



Virginia Commonwealth University
VCU Scholars Compass

Theses and Dissertations

Graduate School

2020

Systematic Model-based Design Assurance and Property-based Fault Injection for Safety Critical Digital Systems

Athira Varma Jayakumar

Follow this and additional works at: <https://scholarscompass.vcu.edu/etd>



Part of the [Computational Engineering Commons](#), [Computer and Systems Architecture Commons](#), [Digital Circuits Commons](#), and the [VLSI and Circuits, Embedded and Hardware Systems Commons](#)

© The Author

Downloaded from

<https://scholarscompass.vcu.edu/etd/6239>

This Thesis is brought to you for free and open access by the Graduate School at VCU Scholars Compass. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of VCU Scholars Compass. For more information, please contact libcompass@vcu.edu.

Systematic Model-based Design Assurance and Property-based Fault Injection for Safety Critical Digital Systems

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science
in Engineering with a concentration in Electrical and Computer Engineering
at Virginia Commonwealth University

by

ATHIRA VARMA JAYAKUMAR

Bachelor of Technology, Electronics & Communication Engineering,
Cochin University of Science and Technology, India, 2009

Director: Dr. CARL R ELKS

Associate Professor, Department of Electrical and Computer Engineering

Virginia Commonwealth University

Richmond, Virginia

May, 2020

Acknowledgement

First and foremost I would like to express my sincere gratitude to my advisor Dr. Carl R Elks, for offering me an excellent opportunity to work in his lab as Graduate assistant which helped me gain a lot of knowledge and valuable experience and helped to cultivate in me the talent of conducting scientific research. His immense knowledge, guidance and motivation helped me achieve the goals in my research and to complete my thesis successfully. He has always been very supportive and understanding and I am extremely blessed to have him as my advisor. I also sincerely thank Dr Ashraf Tantawy and Dr Alen Docef for serving on my thesis committee and providing me valuable feedback and support.

I would like to thank Department of Energy, Division of Advanced Sensors and Instrumentation (DOE-ASI) and Electric Power Research Institute (EPRI) who funded the SymPLe architecture project, which is the representative architecture used for the study presented in this thesis manuscript. I would also like to wholeheartedly thank the Project PI, Matt Gibson of EPRI for all his support during my work in SymPLe project.

I would like to extend my gratitude to all my colleagues in the SymPLe architecture project Smitha Gautham, Richard Hite, Christopher Deloglos and Dr Ashraf Tantawy for all the stimulating discussions we had together, valuable suggestions on my work and all the help you granted me. I would like to express special thanks to Smitha Gautham for becoming a very close friend of mine and for making my stay here at VCU a homely and pleasant experience.

I would also like to express my sincere thanks to Ken Thomas of Idaho National Laboratories, Department of Energies for providing me an opportunity to be part of the valuable study on Bounded Exhaustive Testing to address Software common cause failures. I am also privileged to have got a chance to collaborate with Dr. Richard D Kuhn and Dr Raghu N Kacker of National Institute of Standards and Technology (NIST). I am grateful for your guidance and support on the study on combinatorial testing. I would like to thank my teammates Aidan Collins, Brandon Simon and Anastasios Karles for helping me in my work in the Bounded Exhaustive testing project.

I am grateful to all my teachers for the wisdom they shared, their guidance and blessings. I would also like to extend immense gratitude to my dear parents Kala and Jayakumar for raising me with love, providing me a wonderful childhood and education and constantly supporting and encouraging me. I would not have been able to reach where I am today, without their support and blessings. I would also like to sincerely thank my parents-in-law Geetha and Vijayakumar whose blessings, constant prayers and encouragement helped me to successfully complete my studies. I would like to thank my late paternal grandparents Thankamani and Kerala Varma for all their blessings and unconditional love they showered on me. I would also like to thank my maternal grandparents Bhadramani and Rama Varma who continues to pray for me and offer me endless love. I also thank my sisters Aswathy and Sangeetha who have always been there for me and praying for all success and happiness in my life. I would also like to thank my brothers-in-law Ajithnath and Anand for all their good wishes.

Last but never the least, I would like to thank my dear husband Jayakrishnan and my loving daughter Gauri for all their support and help during my studies. They had been very understanding to offer me all the time I needed to work on research, prepare for my exams and work on assignments. I am extremely grateful to my loving and caring husband for always encouraging me and wholeheartedly wishing for my excellence in any endeavor I take over.

Table of Contents

| | |
|--|----|
| List of Figures | 6 |
| List of Tables | 8 |
| List of Abbreviations | 9 |
| Abstract | 11 |
| Chapter 1 Introduction | 13 |
| 1.1 Motivation | 13 |
| 1.2 Complexity in Software Based Systems | 14 |
| 1.3 Early Detection of Systematic Failures | 15 |
| 1.4 The Role of Standards | 15 |
| 1.5 Summary | 16 |
| 1.6 Outline | 17 |
| Chapter 2 Background | 18 |
| 2.1 Concepts of Dependability | 18 |
| 2.2 Impairments or Threats to Dependability | 19 |
| 2.3 Important Principles for Safety Critical Systems | 22 |
| 2.4 Fault Avoidance and Fault Tolerance Concepts | 23 |
| 2.5 Fault Injection | 24 |
| 2.6 Contributions | 26 |
| Chapter 3 Related Work | 27 |
| Chapter 4 Overview of the Representative System | 30 |
| Chapter 5 Model Based Engineering and Design Assurance | 33 |
| 5.1 Model-based Engineering and Design | 33 |
| 5.2 Model Based Design Assurance | 35 |
| 5.3 Choice of MBD Tool | 36 |
| 5.4 Development of a model based design assurance workflow | 37 |
| 5.5 Model Testing | 43 |
| 5.5.1 Hierarchy of Model Testing | 45 |
| 5.5.2 Simulink Tool Major Features to assist model testing | 46 |
| 5.6 Test Oracle Specification | 49 |
| 5.7 Test Inputs Specification | 50 |
| 5.7.1 Sequence Based Testing | 51 |

| | |
|---|-----|
| 5.7.2 Boundary Value Combinatorial testing:..... | 56 |
| 5.8 Test Execution | 58 |
| 5.9 Test Results Analysis in MBD..... | 59 |
| 5.9.1 Analysis using Logic Analyzer | 59 |
| 5.9.2 Analysis using Animations | 63 |
| 5.9.3 Analysis using State Sequences | 64 |
| 5.9.4 Analysis using Single step Execution | 66 |
| 5.10 Integration Testing | 68 |
| 5.11 System Testing..... | 70 |
| Chapter 6 Model Coverage Analysis Workflow..... | 73 |
| 6.1 Functional Coverage | 74 |
| 6.2 Structural Coverage | 76 |
| 6.3 Reasons for Low Model Coverage..... | 80 |
| Chapter 7 Static Verification | 82 |
| 7.1 Static Conformance to IEC 61508 | 82 |
| 7.2 Design Error Detection | 84 |
| 7.2.1 Example: Dead Logic Detection | 85 |
| Chapter 8 Findings on Model-based Design Assurance | 86 |
| 8.1 Design Flaws detected during Model based V&V..... | 86 |
| 8.2 Lessons Learnt during the Model-based Design Assurance workflow..... | 88 |
| Chapter 9 Model Based Fault Injection | 92 |
| 9.1 Introduction..... | 92 |
| 9.2 Background | 92 |
| 9.3 Classical Fault injection and Strategic Fault Injection..... | 93 |
| 9.4 Strategic Fault Injection..... | 97 |
| 9.5 Introduction to model checking | 98 |
| 9.5.1 Property Coverage: Do I have all of the Properties | 100 |
| 9.6 Model Based Fault Injection using Property Proving | 101 |
| 9.6.1 Challenges with Property based fault injection..... | 105 |
| 9.6.2 Classical Fault Injection vs Property based fault injection. | 106 |
| 9.6.3 Property Based fault injection - Fault-Input-State space Coverage | 110 |
| 9.7 Design and Implementation of a Comprehensive Fault Injection Framework in Simulink..... | 112 |
| 9.7.1 Faults and Fault Model | 112 |
| 9.7.2 Saboteurs..... | 113 |

| | |
|--|-----|
| 9.7.3 Automated Saboteur insertion in model..... | 117 |
| 9.8 Application of Property Based Fault Injection..... | 119 |
| 9.8.1 Use case 1- Verifying Failure Semantics of SymPLe Function Blocks..... | 120 |
| 9.8.2 Use case 2 - Verifying Timeout functionality during SymPLe Function Blocks Execution ... | 124 |
| 9.8.3 Efficiency Comparison between Classical and Property-Based FI..... | 126 |
| Chapter 10 Conclusions and Future Work..... | 128 |
| References..... | 129 |
| Appendix..... | 134 |

List of Figures

| | |
|---|----|
| Figure 1: Decision Effectiveness during life cycle (adapted from [9])..... | 15 |
| Figure 2: Taxonomy of Dependability[14] | 19 |
| Figure 3: Three Universe Model [2] | 20 |
| Figure 4: The SymPLe Architecture [42] | 31 |
| Figure 5: Fault Tolerance Approach for SymPLe [51] | 32 |
| Figure 6: Model-based Design Verification Workflow [54]..... | 36 |
| Figure 7: V&V Workflow for SymPLe architecture..... | 40 |
| Figure 8: Model Testing Hierarchy as applied to SymPLe architecture | 46 |
| Figure 9 : Test Manager Interface [54] | 47 |
| Figure 10: Graphical representation of test assessment results and design signals in Test Manager | 47 |
| Figure 11: Test Harness and Design Model interaction [54] | 48 |
| Figure 12: Simulation, Baseline and Equivalence Testing on Models [54]..... | 50 |
| Figure 13: Local Sequencer Top level State Machine and sub-states in WRITE state..... | 52 |
| Figure 14: Assigning TestCase parameter value in each Testcase in Test Manager | 53 |
| Figure 15: Local Sequencer Test Harness..... | 54 |
| Figure 16: Test Case Description in Test Manager..... | 54 |
| Figure 17: Test Input Sequences in the Test Sequence Block | 55 |
| Figure 18: Verify statements in Test Assessment Block | 55 |
| Figure 19: Feeding Test Inputs and Expected Outputs in an Array format within the M script..... | 57 |
| Figure 20: Test Results generated by M Script - Inputs, Expected & Actual outputs for testcases in Tabular format | 58 |
| Figure 21: Test Results Summary in Test Manager..... | 58 |
| Figure 22: Graphical plot of Verify statement results with respect to test execution time in Test Manager | 60 |
| Figure 23: Analyzing Local Sequencer input and output signals in Logic Analyzer..... | 60 |
| Figure 24: ERROR State in Local Sequencer State Machine | 61 |
| Figure 25: Local Sequencer State Chart Settings..... | 62 |
| Figure 26: FB Controller Test Inputs in the Logic Analyzer and Animations in FB Controller State Machine | 64 |
| Figure 27: (i) State Sequence View with Bug in FB Controller, (ii) State Sequence View with Bug fixed in FB Controller | 65 |
| Figure 28: Single step execution in subtractor function block..... | 66 |
| Figure 29: Error Codes implemented in SymPLe | 67 |
| Figure 30: Datatype Error Detection..... | 67 |
| Figure 31: Deadlock Scenario detected in Integration testing | 68 |
| Figure 32: Integration Test Failure: Local Sequencer state sequence view | 69 |
| Figure 33: Local Sequencer State sequence view after Design fix | 69 |
| Figure 34: PID Controller and Plant Closed Loop Test Environment. | 70 |
| Figure 35: Equivalence Test Environment for PID Application on SymPLe architecture | 71 |
| Figure 36: Graphical plots of PID outputs from SymPLe PID and the two reference PIDs in Simulink Data Inspector | 72 |
| Figure 37: Functional and Structural Coverage [54]..... | 73 |

| | |
|---|-----|
| Figure 38: Requirements Editor view showing Implementation and Verification status and Forward traceability links from Requirements to testcases..... | 75 |
| Figure 39: Test Manager view showing Backward Traceability links from Testcases to Requirements ... | 76 |
| Figure 40: MC/DC Coverage [54] | 78 |
| Figure 41: MC/DC Coverage highlighted in the model..... | 79 |
| Figure 42: Model Coverage Analysis summary | 80 |
| Figure 43: Low coverage due to over specified design..... | 81 |
| Figure 44: Static analysis checks for IEC 61508 compliance..... | 83 |
| Figure 45: Dead Logic Detected during static Design error detection on Models..... | 85 |
| Figure 46: Design faults found in SymPLe architecture by V&V process | 89 |
| Figure 47: Non Linear Model based verification process..... | 91 |
| Figure 48: Fault Space | 94 |
| Figure 49: Model Checking | 98 |
| Figure 50: Uncovered state-output space points due to missing properties..... | 101 |
| Figure 51: Simulink Design Verifier Blocks: Proof Objectives, Constraints and Temporal Operators [90] | 103 |
| Figure 52: Fault Injection with Property Proving | 105 |
| Figure 53: Classical Fault Injection vs Property Based Fault Injection..... | 106 |
| Figure 54: Classical Fault Injection Workflow..... | 107 |
| Figure 55: Property Based Fault Injection Workflow..... | 110 |
| Figure 56: Property Based fault injection - Fault-Input-State space Coverage..... | 111 |
| Figure 57: Fault Model Taxonomy | 113 |
| Figure 58: Fault Injection Control Signals..... | 114 |
| Figure 59: Saboteur Masked Subsystem..... | 115 |
| Figure 60: Saboteur Location Parameter | 115 |
| Figure 61: Inside the Saboteur subsystem: Level 1 | 116 |
| Figure 62: Inside the Saboteur subsystem: Level 2 | 116 |
| Figure 63: Saboteur Inserted model..... | 118 |
| Figure 64: Hardware Redundancy in Function Blocks [48] | 119 |
| Figure 65: Property for proving hardware redundancy of Function Blocks | 122 |
| Figure 66: Input data equivalence..... | 122 |
| Figure 67: Funcblock selection..... | 122 |
| Figure 68: Fault Injection Control Signals..... | 123 |
| Figure 69: Timeout Detection Feature Property | 126 |

List of Tables

| | |
|---|-----|
| Table 1: Implemented Function Blocks in SymPLe [51]..... | 32 |
| Table 2: Table showing IEC 61508 certified Mathworks Products | 37 |
| Table 3: V&V workflow phases to IEC 61508 Requirements..... | 39 |
| Table 4: Faults Found during the presented Model-based Design Assurance Process | 86 |
| Table 5: Function Block Error Detection Capabilities [51] | 120 |
| Table 6: Function Block Fault Injection Campaign Results | 123 |

Publications

(Primary) Achieving Verifiable and High Integrity Instrumentation and Control Systems through Complexity Awareness and Constrained Design. No. 15-8044. Electric Power Research Institute (EPRI), 2019.

(Primary) Preliminary Results of a Bounded Exhaustive Testing Study for Software in Embedded Digital Devices in Nuclear Power Applications: Light Water Reactor Sustainability Program Idaho National Laboratory U.S. Department of Energy, Sep 2019

(Secondary) Lessons and Experiences Learned Applying Model Based Engineering to Safety Critical FPGA Designs. October 2018 Conference: 11th International Workshop on the Application of FPGAs in NPPs

In preparation

(1st Author) Property-Based Fault Injection: A Novel Approach to Model-Based Fault Injection for Safety Critical Systems, IMBSA 2020

(1st Author) On the Application of Systematic t-way Software Testing for Safety Critical Embedded Digital Devices in Nuclear Power

(2nd Author) Findings and experiences on applying IEC 61508 compliant model-based verification to an FPGA based system for Nuclear Industry, IMBSA 2020

(2nd Author) Design and Assessment of multilevel Runtime Verification framework using model-based engineering

(2nd Author) Design and Realization of Runtime Verification Monitors using Model-based Engineering and Event Calculus

List of Abbreviations

| | |
|-------|--|
| CPS | Cyber Physical Systems |
| MBD | Model Based Design |
| IC | Integrated Circuit |
| IEC | International Electrotechnical Commission |
| FPGA | Field Programmable Gate Array |
| IP | Internet Protocol |
| LS | Local Sequencer |
| GS | Global Sequencer |
| FB | Function Block |
| HW | Hardware |
| SW | Software |
| MBDE | Model Based Design Engineering |
| DV | Design Verifier |
| ISO | International Organization for Standardization |
| V&V | Verification & Validation |
| MIL | Model-In-Loop |
| RTL | Register Transfer Level |
| SEU | Single Event Upset |
| HDL | Hardware Description Language |
| VHDL | Very High Speed Integrated Circuit Hardware Description Language |
| FI | Fault Injection |
| LTL | Linear Temporal Logic |
| MTL | Metric Temporal Logic |
| FMEA | Failure Mode and Effects Analysis |
| XML | Extensible Markup Language |
| MC/DC | Modified Condition / Decision Coverage |
| I&C | Instrumentation & Control |
| PID | Proportional Integral Derivative |
| SIL | Safety Integrity Level |

| | |
|-------|---|
| CLB | Configurable Logic Block |
| UML | Unified Modeling Language |
| FBD | Function Block Diagram |
| PLC | Programmable Logic Controller |
| FTA | Fault Tree Analysis |
| STAMP | System-Theoretic Accident Model and Processes |
| STPA | Systems Theoretic Process Analysis |
| ABVFI | Assertion Based Verification Fault Injection |
| BDD | Binary Decision Diagrams |
| FIM | Fault Injection Module |
| ASIC | Application-Specific Integrated Circuit |
| MBT | Model Based Testing |
| NPP | Nuclear Power Plant |
| IO | Input Output |
| SIHFT | Software Implemented Hardware Fault Tolerance |
| FDIM | Fault Detection, Isolation and Mitigation |
| SDLC | Software Development Life Cycle |
| GUI | Graphical User Interface |
| MBDA | Model Based Design Assurance |
| SRAM | Static Random-Access Memory |
| ID | Identifier |
| NaN | Not a Number |

Abstract

SYSTEMATIC MODEL-BASED DESIGN ASSURANCE AND PROPERTY-BASED FAULT INJECTION FOR SAFETY CRITICAL DIGITAL SYSTEMS

By Athira Varma Jayakumar, B.Tech.

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science
in Engineering with a concentration in Electrical and Computer Engineering
at Virginia Commonwealth University

Virginia Commonwealth University, 2020

Major Director: Dr. Carl R Elks

Associate Professor, Department of Electrical and Computer Engineering

With advances in sensing, wireless communications, computing, control, and automation technologies, we are witnessing the rapid uptake of Cyber-Physical Systems (CPS) across the spectrum, with a potential economic impact of as much as \$11.1 trillion per year by 2025 in different settings, including connected vehicles, healthcare, energy, manufacturing, smart homes, and smart cities [1]. Many of these applications are safety-critical in nature. Meaning that they depend on the correct and safe execution of software and hardware that are intrinsically subject to faults. These faults can be design faults (Software Faults, Specification faults, etc..) or physically occurring faults (hardware failures, Single event upsets, etc..) . Both types of faults must be addressed in design, development and implementation of these types of critical systems.

As such, several safety critical industries (including automotive, avionics, medical) have widely adopted Model Based Engineering and Design paradigms to address and manage the design assurance processes of these complex CPSs. This thesis studies the application of IEC 61508¹ compliant model based design

¹ IEC 61508 “*Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems (E/E/PE, or E/E/PES)*”, is a basic functional safety standard applicable to all kinds of industry. It defines functional safety as: “part of the overall safety relating to the Equipment Under Control.

assurance methodology on a representative safety critical digital architecture targeted for the Nuclear power generation facilities. The study presents detailed experiences and results to demonstrate the benefits of Model testing in finding design flaws and its relevance to the subsequent verification steps in the workflow.

This thesis extends the model based design assurance workflow in part 1 to include ‘model based fault injection based on property proving’, to study the response of the system to injected faults. To address the impact of physical faults on the digital architecture we develop a novel property based fault injection method which overcomes some of the deficiencies of traditional fault injection methods. The model based fault injection approach presented here is novel in terms of the high efficiency and input/state/fault space exhaustiveness achieved, by utilizing formal verification principles to identify fault activation conditions and prove the fault tolerance features. The fault injection framework facilitates automated integration of fault saboteurs throughout the model to enable exhaustive fault location coverage in the model. The property based fault injection technique we developed has not been reported in the literature to date.

Chapter 1 Introduction

1.1 Motivation

We are witnessing the rapid uptake of Cyber-Physical Systems (CPS) technology across the spectrum, with a potential to impact society in ways that could not be envisioned 15 years ago. It is estimated the economic impact of such technology could be as much as \$11.1 trillion per year by 2025[1]. Cyber-physical systems (CPS) integrate a diverse set of hardware and software components to provide new service capabilities of which some could be of a critical nature. Technologies that enable the realization of CPSs include embedded computing platforms, System on Chip, smart sensors and actuators, IP based software, and wireless communication networks. Examples of their everyday use include energy operation centers, wearable devices, connected healthcare, driver-assisted automobiles, military coordinated operations and homeland security monitoring centers, to name a few. As CPS technology evolves and matures, we, as a society will depend on them more and more for critical services – 911 call centers, water supply and distribution, air space management, etc.

A system is deemed *critical* if the services the system provides to its end-users or environment are important for the end-users or environment to function normally. A system is *safety critical* if the loss of service can lead to hazards that affect the safety of the end-users or the environment. Hazards are the precursors to accidents, outages, catastrophes that include loss of life, damage to the environment and infrastructure. Examples of safety critical systems include commercial aircraft, medical therapy and diagnosing equipment, autonomous driving systems, and nuclear reactor protection systems – among many.

Very often these systems must provide uninterrupted operation to supply critical service even in the presence of faults or become fail-safe until failures are repaired. Redundancy and fault tolerance strategies are used to cope with randomly occurring physical faults and failures [2]. However, faults that arise from design oversights, software coding mistakes or inadequate requirements are called *systemic failures* that can lead to *common cause failures*. Meaning, their unintentional activation can result in complete failure of the system and thus defeat any redundancy and safeguards mechanisms. As an example, in 2014 a 911 regional emergency call routing center in Colorado failed in way that prevented 911 emergency calls to route to their destination. The root cause of the problem was traced to a specification/software flaw responsible for assigning the route codes. The software maxed out at a pre-set

limit; the software literally stopped counting at 40 million calls. As a result, the routing system stopped accepting new calls, leading to a series of cascading failures across 8 states in the 911 infrastructure. For reasons and circumstances like this, detecting and removing systemic design faults are of particular concern in the research community of safety critical systems [3].

1.2 Complexity in Software Based Systems

Computer hardware and its software in the context of critical computing are sometimes called an *embedded computer or device*. An embedded digital device in a car, for example, can be the software (SW) controlled gearbox or an anti-locking brake system computer; that has a function within the overall system – the car. Software controlled embedded devices like microprocessors do not; however, behave like usual physical machine elements. The microprocessor is a general purpose device that when given a program to execute, in effect becomes the machine it is designed (programmed) to be. Simply explained, using a microprocessor we no longer need to build intricate mechanical or analog devices to control, for example, a gearbox controller. We simply need to write down the design of the controller using SW instructions to accomplish the intended goal. These instructions are then made available to the microprocessor which, while executing the instructions, in effect becomes the controller. The immense flexibility of computers and their ability to perform complex tasks efficiently, have led to a great increase in their use, even into critical systems where the consequence of failure could be significant. This immense flexibility has a downside - complexity naturally arises. Complexity of software and hardware design is one of the primary sources of design flaws that can lead to serious system failures in safety critical systems.

As we have observed in recent cases, highly integrated systems can be susceptible to cascading failures leading to accident consequences [4]. The root causes of these failures are very often traced back to design flaws where poorly understood interdependencies between computer control systems and automation result in incorrect and unsafe actions [5]. As complexity increases, design faults are more prone to occur when more interactions make it harder to identify hazard behaviors through traditional software and system testing methods [6]. For these reasons, more powerful design assurance methods and models that capture these complex interactions with assumptions and interactions are much needed at this time.

1.3 Early Detection of Systematic Failures

The critical nature of CPSs requires thorough and effective analysis throughout their lifecycle but especially at the early design development phase. This is supported by several studies where it is estimated that 70-80% of the decisions affecting safety and security are made in the early concept development phase of any project [7] [8]. Figure 1 illustrates the importance of detecting and removing systematic design errors. First, early identification of a design flaw in the lifecycle significantly reduces the cost of remediation (“fixing” the bug) as compared to late lifecycle detection. Secondly, detecting a design flaw early in the lifecycle process removes the risk that the design flaw leads to hazard and endanger lives. Early systematic detection, analysis and mitigation of design flaws are part of the growing technical body of research called Design Assurance Methodologies and Testing.

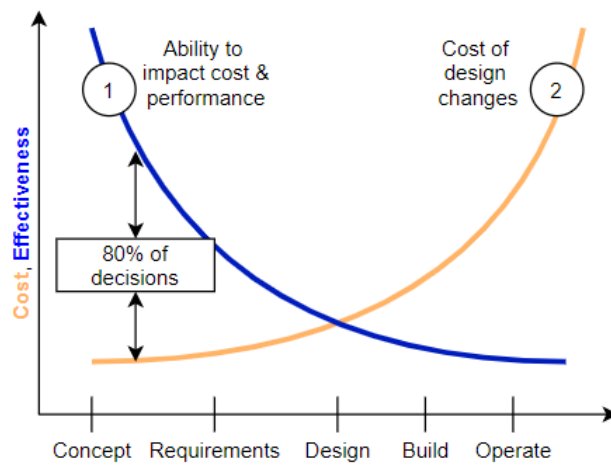


Figure 1: Decision Effectiveness during life cycle (adapted from [9])

1.4 The Role of Standards

With critical embedded systems being increasingly used in society, the safety of the products becomes extremely important as the consequence of failure can cause harm to life and/or environment. Consequently, safety critical systems industries are regulatory and standards driven – meaning products are required to comply with standards and guidelines. Each industry has a unique set of governing safety standards for product software and system development. As example, IEC 61508 is the general International functional safety standard for Electronic/Programmable Electronic Safety related systems.

ISO 26262 is the functional safety standard for automotive, EN 50128 for Railways, IEC 62304 for Medical devices and DO-178B for Airborne Systems. For achieving certification on these safety standards, the product design needs to demonstrate the management of both random and systematic failures of the product. In addition to that, there should be evidences to show that product development follows proven and recommended failure risk reduction processes.

A standard of particular interest to this research is IEC 61508. This standard is important for several reasons; (1) because it is like a “mother” standard, as many other safety critical system standards like ISO 26262, DO-178B, DO-254, IEC 61513 (nuclear 61508) are heavily influenced by 61508-3; (2) its recent consideration by US Nuclear Regulatory Commission and its longtime acceptance by the international nuclear energy community. Additionally, IEC 61508-3 provides guidance on both software and hardware aspects of design assurance which is important for a HW and SW based design.

IEC 61508 defines various levels of Safety Integrity Levels (SIL) for systems with SIL level 4 being considered highest dependable system and SIL level 1 being considered the least. IEC 61508-3 defines the guidelines and processes to be followed at each stage of the software development including requirements, design and verification. IEC 61508 specifies several *normative requirements* that influence the choice of design assurance workflow and tools. These include V model, Bi-directional traceability and use of Formal and Semi formal methods. Forward and backward traceability between all levels of artifacts ranging from requirements, design, code and verification is a mandatory requirement for SIL levels 3 and 4 in IEC 61508. IEC 61508 recommends the usage of “V model” to represent the software development lifecycle. A more detailed discourse on IEC-61508 can be found at [10].

1.5 Summary

Engineering history has revealed that design faults are major causes of catastrophic failures, from the Tacoma Narrows Bridge collapse to the Boeing 737 MCAS accidents [11]. Survey on failure sources of traditional fault tolerant and non-fault tolerant computing systems have found that almost 75% of system failures originated from flaws in early design phase activities that are eventually propagated to software and hardware implementation.[12] With design faults being a bottleneck of systems dependability, effective and comprehensive methods need to be employed during the specification and design and verification of safety critical systems. At this time, there are numerous approaches and methodologies being proffered [13]. This thesis examines two aspects of design assurance for safety critical CPSs; (1) the feasibility, benefits and challenges of using IEC 61508 standard from a *model-based verification*

perspective, and (2) the development and application of a novel fault injection technique that overcomes problems with traditional fault injection. Specifically, we address the physical faults problem by building a model-based fault injection framework based on property proving to analyze the impact of physical faults on system models and verify the fault tolerance features very early in the design phase. The overall goal of this thesis is to investigate, develop, and collect credible evidence on the efficacy of model based design assurance in the context of IEC 61508. A further contribution of this thesis is the novelty of developing model based design assurance workflows for FPGA based systems which have not been widely reported in the literature.

1.6 Outline

This thesis is organized as follows. Chapter 2 gives background information on this research work; discusses dependability concepts and safety critical systems principles, and states the contributions of this thesis. Chapter 3 highlights related work in the field of model based design assurance and fault injection. Chapter 4 gives brief overview of the architecture and components of the representative system that is subjected to this experimental study on model based design assurance. Initial part of chapter 5 describes the basic concepts of Model based engineering and the selection of MBD tools. Further, the chapter presents the model based design assurance workflow developed for IEC 61508 compliance and describes the Model testing phase and experimental results in detail. Chapter 6 talks about the model coverage analysis phase and the implications of low model coverage. Chapter 7 provides details on the static analysis phase in the presented V&V workflow. Chapter 8 summarizes the findings and lessons learned during the application of the Model based design assurance process. Chapter 9 discusses the proposed approach of property-based fault injection at the model-level and presents the experimental results. Finally, Chapter 10 gives the Conclusions and Future work.

Chapter 2 Background

This chapter gives an overview of dependability concepts to place the work in context. It also introduces common definitions, terms, and relationships that are a part of dependability theory. The later part of this chapter gives a brief background on fault injection, its importance to design assurance and finally the major contributions of this work.

2.1 Concepts of Dependability

Dependable systems concepts and technology have evolved over the past 40 years from a large community of researchers and practitioners. A significant contribution to dependability concepts and theory is the work by Avizienis, Laprie, and Randell [14]. This work is widely recognized as a standard for understanding the concepts of critical system attributes. This thesis adopts [14] concepts of dependability to provide consistent terminology when discussing faults, errors, and failures with respect to safety critical systems. For a more in depth review of dependability concepts, [14] paper provides a detailed appreciation of the topic. We say *dependable system* has the ability to provide its intended, expected and agreed upon functions, behavior, and operations in a correct timely manner [14].

In dependability theory, a *system* is defined as an interconnected set of elements that is coherently organized in a way that achieves objective or purpose. These elements can be other systems, components, including hardware, software, humans, and the physical environment. The border between the system and its environment is the *system boundary or common interface*. The *service* delivered by a system is its behavior as it is perceived by its user(s) – which could be other systems, humans, and physical control. [15]

V

The common definition of *Dependability* is “Dependability is the ability of a system to deliver service that can justifiably be trusted”[14]. Another alternate definition of dependability is, “Dependability is the ability of a system to operate in such a way that the frequency and severity of the service failures happening are acceptable to the user.”[14]. The second is more accepted than the first.

The *attributes* of dependable systems are the primary means by which the quantitative and qualitative requirements of a system are specified. Figure 2 shows the taxonomy of dependability attributes. The following are some basic terms and concepts related to dependable system attributes:

Definition 1: *Reliability*, a conditional probability that the system will perform correctly throughout the interval $[t_0, t]$, given the system was performing correctly at time t_0 [2], which is related to the continuity of service.

Definition 2: *Availability*, a probability that a system is operating correctly and is available to perform its functions at the instant time, t [2], which is related to readiness for usage.

Definition 3: *Safety*, a probability that a system will either perform its functions correctly or will discontinue its functions in a manner that does not disrupt the operation of other systems or compromise the safety of any people associated with the system [2], which is related to the non-occurrence of catastrophic consequences on the environment.

Definition 4: *Integrity*: absence of improper system alterations [2].

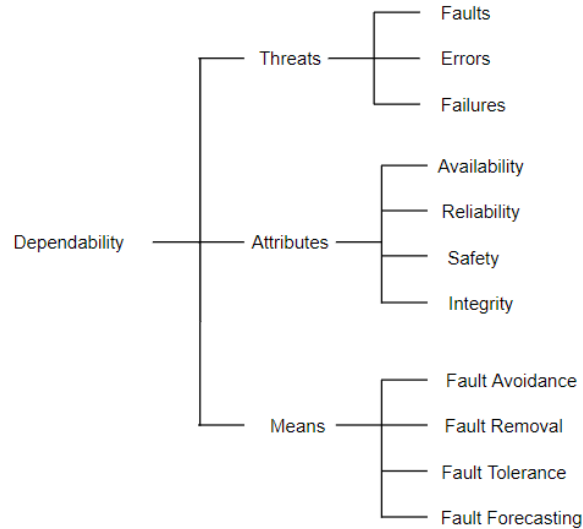


Figure 2: Taxonomy of Dependability[14]

2.2 Impairments or Threats to Dependability

Faults, errors, and failures affect the ability of a system to deliver its dependability attributes (e.g., safety, reliability, performance, etc.). Hence, they are called the *impairments* or *threats* of dependability. Correct service is delivered when the service accurately reflects the system function. A *service failure*, abbreviated here to *failure*, is an event that occurs when the delivered service deviates from correct service. A failure is either because the service does not comply with the functional specification, or because this specification did not adequately describe the system function. A failure is a transition from

correct service to incorrect service (i.e., not implementing the system function). The deviation from correct service may assume different forms that are called failure modes.

In digital systems, delivered service is dependent on sequence of execution states, a service failure means that at least one (or more) states of the system deviates from the correct service state. This deviation is called an *error*. Error propagation occurs when an error is successively transformed into other errors through execution of instructions, program states, and functions on a digital system, (e.g., errors from function A propagate to function B when it receives information from component A).

The cause of an error is called a *fault*. Faults can be internal or external to a system. Given a fault is present in a system, the fault turns active when it manifests into an *error*; otherwise, it is latent. Fault activation occurs when various activation factors align to initiate the dormant fault – these include inputs, modes of operation, program execution, etc.

Implicit in the definitions of the above terms is a cause-effect relationship. The well-known 3-universe model depicted in Figure 3 shows the relationship between faults, errors, and failures [2]. Faults cause errors, and errors may propagate to the system interface to cause service failures.

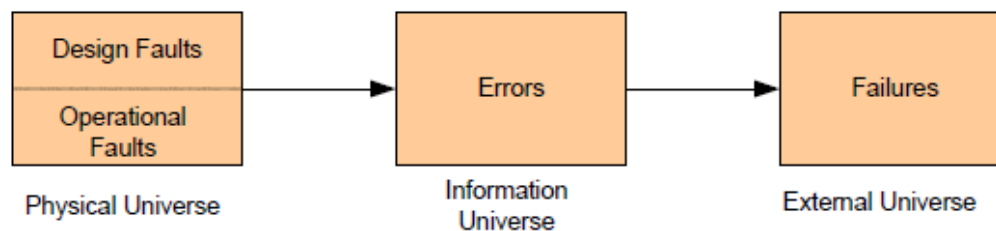


Figure 3: Three Universe Model [2]

The physical universe depicts where faults can originate. *Operational faults* are associated with the *use* phase of the system. During the *use* phase, the system interacts with its environment and may be adversely affected by faults originating in it. These types of faults include a wide variety of faults. Such as;

- Faults from naturally occurring phenomena – HW faults, semiconductor faults, power faults, connector faults, etc.

- Faults introduced from improper operator use – improper commands, improper configuration, omission faults, etc.
- Faults introduced from malicious intruders – cyber-attacks, theft of information, exploitation, etc.

Design faults occur in the development phase of the system. The development phase includes all activities from requirements to testing to implementation to deployment. During the development phase, the system design process successively refines the system from requirements and specification to hardware and software implementations. During the development process, design and development faults may be introduced.

Design faults can arise from a number of sources. A few are;

- Fault arising from mistakes - misinterpretations of requirements, ambiguity in specification language, improper implementation of design, etc.
- Faults arising from omissions – Requirements that were overlooked, incomplete requirements or specifications, assumptions not stated, functionality left out of implementations, etc.
- Faults arising from improper commission – Requirement that is not needed, Functionality that is present but not needed for the application, redundant functionality, dead logic, implementation has functionality that was not intended, etc.

Physical faults on the other hand are faults that occur due to adverse physical phenomena [16] and affect the operation of the system, although the design of the system could be flawless. Manufacturing defects or hardware damages like permanent open or short circuits that cause the electronic component to stop functioning or function incorrectly are examples of Physical faults. Cracking and critical failure of the semiconductor devices happening as a result of electrical, thermal or mechanical stress beyond the device's maximum ratings could be causes of permanent physical faults. Power fluctuations, switching transients, soft errors arising due to electromagnetic interferences and cosmic radiations are other major reasons for transient physical faults. Single or multi event upset is the change in state of memory or logic elements in a micro electronic device when they are struck by high energy charged particles. They are classified as soft errors as they are non-destructive and get fixed when the flipped bits get back to the previous state. [17] Alpha particles which are essentially particles identical to a helium-4 nucleus released from thorium and uranium impurities in the packing material of the device are one of the sources of the soft errors [18]. Cosmic rays which are a major source of soft errors, highly threatening to the aerial

devices, are high energy particles originating in space. The cosmic rays when entering the earth's atmosphere causes a series of cascaded particle showers thereby generating high speed neutrons. The alpha particles and neutrons when passing through the layers of semiconductor device material are capable enough to create an ionization path with free electrons and holes causing change of state of storage elements within the device. [19] Another category of transient physical faults is the delay fault. Delay faults represent faults that arise due to propagation and transition delays at the gate level during power supply disturbances and single event transients. Delay faults at a component in the hardware do not make the component logically incorrect, but instead delays its response causing a timing error. Another set of physical faults are intermittent faults that appears at irregular intervals possibly due to a loose connection in the circuit or unstable hardware and software states or intermittent power transients.

Interaction faults are the class of faults that occur due to faults in the human – machine interactions. These normally happen when right procedures and good practices are not followed during the operation or maintenance of the devices. They could be accidental or deliberate and of malicious or non-malicious intent. As interaction faults are human-made faults, they are usually not considered when studying system centric faults as in other studies [16]. The same approach is taken in this work by addressing only the design and physical faults in the fault space.

2.3 Important Principles for Safety Critical Systems

In the context of safety critical systems, the goal is to preserve safety. Accordingly, failures associated with digital systems that lead to hazardous conditions must be extremely improbable². Digital systems without any additional protection against failures are most likely to fail in ways that are unacceptable to the overall safety of the system. Thus, a defensive fault tolerant strategy must be employed to do one or two actions upon the occurrence of erroneous behavior associated with the digital system. These are:

- 1 **A fail-safe action:** Upon detection of an error that could lead to failure, *force* (by some defensive measure) the digital system to discontinue its operation in manner that does not compromise the safety requirements of the overall system [20].

² The term “extremely improbable” is dependent on the application requirements which vary from application to application. For example, the requirement for digital flight control systems in civilian air transports is 10⁻⁹ failures per 10 hour flight –which is extremely improbable.

- 2 **A fail-operational action:** Upon detection of error that could lead to failure, *overcome* (by some corrective action) the erroneous condition such that the operation of the digital system is able to continue according to its specified safety and functional requirements [20].

Fail-safe and fail-operational actions are principle goals for safety critical systems. These actions are not inherent in most digital systems; the system must be designed to behave as a fail-safe or fail-operational system. Traditionally, fault detection and/or fault tolerance mechanisms are added to the fault intolerant design to detect and possibly correct the effects of faults during the operation of the system. There are four main approaches as described by [14] that are a *means* to realize more dependable components:

- 1 **Fault avoidance** means to prevent the occurrence or introduction of faults. How to avoid faults by design, i.e., try to build the system correctly from the beginning.
- 2 **Fault tolerance** means to avoid service failures in the presence of faults.
- 3 **Fault removal** means to reduce the number and severity of faults. How to reduce; by testing and verification to find the presence of faults
- 4 **Fault forecasting** means to estimate the present number, the future incidence, and the likely consequences of faults.

This thesis is focused on methods related to fault avoidance, fault removal, and testing fault tolerance mechanisms via fault injection.

2.4 Fault Avoidance and Fault Tolerance Concepts

Fault avoidance refers to methods and practices used for producing a system or component that are, by design, free of design flaws to as “low as reasonably practicable”. Fault avoidance strives to prevent faults from being introduced into the systems by; (1) applying rigorous design and specification processes, and/or (2) taking necessary quality control measures during the design, implementation, and manufacturing of the system. As stated earlier, design faults are one of the major causes of failures in digital systems (and software intensive systems). Examples of fault avoidance methods are following safe programming practices, performing rigorous coverage based SW testing [21], formal specification and verification [22] and extensive reviews on the system. Quality control during manufacturing involves selection of defect free components, strong production process disciplines and quality inspection checks of products. While design faults can be avoided using rigorous design and verification process, physical faults that occur randomly during the operation of the system can be only handled with well-formed fault detection and handling techniques for the system.

While fault avoidance is a process-oriented concept, *fault tolerance* is a product-oriented concept. Fault tolerance can be defined as the “property of a system to detect the presence of malfunctioning resources, or faults within or external to the system and continue to operate normally”[23]. Fault tolerance almost always involves the use of redundancy either at the system, hardware and software level to detect and overcome the effects of faults and thereby prevent system failures. A good introduction and detailed discourse on fault tolerance techniques can be found in [23] [24].

2.5 Fault Injection

In order to study the effect of physical faults on systems, the technique of fault injection was introduced in the early 1970s. Fault injection is used to verify the fault tolerance capabilities of a system by deliberately injecting faults into the system. The types of faults injected into the system depend on the fault model used. A *fault model* is a representation that captures the behavioral and temporal aspects of faults the system is expected to experience in its operational life. The physical faults listed in the impairments and threats section of these are often studied to produce a representative fault model. The fault model is then used with fault injection techniques to inject the faults into the target system. There are a wide variety of fault injection techniques with advantages and disadvantages with respect to properties like reachability, controllability, repeatability, intrusiveness and observability. Several references provide excellent comparative overview of various fault injection techniques [25], [26].

At the broadest stance, fault injection can be broken down into three types: Physical hardware fault injection, Software based fault injection, and Simulation based fault injection.

Hardware Fault injection is a physical fault injection technique in which the faults are directly injected into an IC or device. The hardware injection could be with or without direct contact to the target hardware. Probing the pins on the target hardware is one of the easiest hardware fault injection by which the current or voltage at the pins are varied thereby causing a stuck at high or stuck at GND fault. Bridging faults are induced by connecting the probe across two or more pins. Fault injection without contact is used to inject single bit and multi bit upsets by exposing the target hardware to high energy ions radiations or electromagnetic fields. This method provides a high level of accessibility to corrupt storage elements in parts of the circuit which may be elusive by other means.

In Software fault injection, faults are induced within the software through instrumented fault injection software. The basic idea of Software fault injection is to corrupt memory and register location of the processor to emulate bit flip faults and transient faults. In practice two types of Software fault injection are often used: Software Implemented Fault Injection (SWIFI), and Compile time fault injection. SWIFI is a runtime fault injection method where a special interrupt service routines are used to “momentarily” halt the execution of code, and insert corruptions into memory and register spaces. This allows injecting faults in all levels of software including application, software driver code and operating systems. On the other hand, compile time fault injection is where different faults are inserted into the source code, compiled and executed on the target processor. Compile time fault injection can be compared to mutation testing where the target software instructions are altered to change the behavior of software thereby simulating either a physical fault or a software fault.

Simulation Based Fault injection often uses a high fidelity model of the system and simulates the code running on the model. This technique can inject fault onto a hardware or software simulation of the system model instead of directly injecting them on the actual prototype. This gives a lot of advantage, the main being that it helps in performing system failure analysis at an earlier stage in the development process. Examples of simulation tools to support simulation based fault injection are OVPsim, Simics and QEMU [27] [28]. Hardware simulations could be a VHDL or Verilog representation of the system as well. One approach to simulated FI is with the simulation model modification. With fault injection modules called *saboteurs* being inserted in the signals to be tested, the tester is able to control the activation and duration of the fault precisely and the effect the fault can be monitored on external outputs as well as on the internal signals. A second approach to simulated FI is through simulator commands where *built in* simulator commands are used to modify signal and variable values.[29]

Most fault injection methods are applied at latter stages of lifecycle development where hardware and software implementation are generally mature. The disadvantage to applying fault injection at latter lifecycle phases is twofold: Cost of changing the design to fix problems, and coverage of the fault injection. In keeping with the themes of this thesis, our interest is to explore fault injection concepts and methods that can be applied before code and hardware is developed – the so called *functional model level*. The ability to perform early systematic fault injection and propagate the results of the fault injection results into latter design decision phases can have a significant impact on design assurance.

2.6 Contributions

The first part of this research work is focused on systematic development and application of fault avoidance and fault removal methods with respect to safety critical systems, specifically model based design assurance and testing methods. This thesis addresses the challenges of uncovering design faults of a digital architecture by applying model based design and verification principles in a strict IEC 61508 context. The representative system considered for this study is targeted for SIL level 3 and 4 – the most stringent in safety applications. Our approach investigates design assurance with rigorous model testing, formal verification of critical system properties using property proving and static analysis to find design faults and IEC 61508 compliance issues.

The second phase is aimed at accessing *fault tolerance* performance of the representative system due to the impact of physical faults on the system through an innovative fault injection method. A novel methodology called *property based fault injection* is introduced that provides an exhaustive coverage of the fault, input and state space of system components - with respect to fault tolerance and safety properties.

The contributions of this work are:

- Formulation, development and evaluation of a comprehensive end to end design assurance workflow that spans from model to code to the hardware implementation, to support IEC 61508 safety standard.
- Gathered evidence for demonstrating the effectiveness and capacity of Model Testing and model coverage methodology to find a diverse set of design flaws.
- Formulated and developed a highly efficient concept of Property Based fault injection at the model level that is more effective and productive with respect to traditional fault injection in its completeness of input, state and fault space coverage.
- Developed and implemented a fault injection framework that facilitates *automated fault injection* to enable high coverage fault injection in system models.

Chapter 3 Related Work

This chapter highlights several previous research and work on the Model-based design assurance and V&V activities aimed at achieving a high level of dependability for products in safety critical industries. It also presents related work in the literature on model-based or formal methods based fault injection techniques.

There has been several surveys about current state-of-the practice in the use and the assessment of Model based engineering in several industry domains, like in space and aeronautics [30] and in general embedded systems [31]. Murugesan et al in [32] presents a study on the application of model-based design and verification to medical device software development. The work describes the use of Simulink for describing continuous time control system models, Stateflow FSMs to model the mode logic behavior and Architecture Analysis and Description Language to represent the architectural models for a medical CPS. The authors also describe about the Assume guarantee approach used to verify if the architecture meets requirements and formal verification using Design Verifier to verify if the detailed behavioral model meets safety properties. Bhatt et al in [33] describes the lessons learnt and challenges faced during the adaptation of Model based design and assurance to avionics applications for compliance to DO-178B standard. The authors give an interesting comparison of the traditional vs MBD design assurance environments. Bringmann and Kramer in [34] studied the challenges involved in model based testing and introduces a new approach of MBT called the Time Partition Testing which can graphically represent test models for complex and closed loop real time environments in automotive domain. Beine's paper [35] gives a reference workflow for model based development of safety critical systems in order to achieve highest safety integrity levels as per IEC 61508 or ISO 26262 standards. The author discusses the contribution of a detailed workflow in achieving safety certification of the software tools using example of Targetlink code generation tool. Mathworks papers [36] and [37] describes the IEC 61508 and ISO 26262 compliant V&V workflows in the context of Mathworks Simulink toolchain. Marrone et al in [38] presents an approach to automatically generate formal models and test cases from high level models of the system that contain all the information necessary to the V&V purposes. This paper introduces a Model-based process driven by UML profiles and model transformations that map structural and behavioral UML diagrams to formal models and applied onto a railway control system. It also describes a software architecture supporting testing and verification activities. The study [39] proposes an efficient technique for verification and validation of automotive software in the context of ISO-26262 compliance by applying fault injection techniques combined with mutation testing approach at the model level. The

improvements include such aspects as identification of safety related defects early in the development process thus providing enough time to remove the defects.

In addition to the popularity of MBD gained in automotive, aerospace and industrial equipment industries nuclear industry has also started showing interest towards application of MBD assurance for safety critical applications. The use of the Simulation Validation Test Tool SIVAT for validating the safety critical software in Areva Teleperm TXS I&C system at the Oconee NPP [40] was one of the first model based verification attempts for nuclear energy I&C. There are several model based design verification tools used in different industries. One such tool for the Nuclear industry applications is the Nude 2.0 developed by Konkuk University and KAERI. Nude 2.0 [41] is a model based system development environment that provides automatic code generation (C, FBD, Verilog, VHDL), verification and safety analysis environment for PLC/FPGA based safety-critical digital I&Cs. It offers support for verification techniques like simulation, model-checking, equivalence checking and safety analysis techniques like STAMP/STPA and FTA.

There are several surveys [26] [42] [43] that study the state-of-art techniques in different categories of conventional fault injection - hardware-based, software-based, simulation-based and emulation-based approaches. In the last decade, FI community has started gaining interest on the application of formal methods to fault injection and to the analysis of fault tolerance mechanisms. Few works that extend formal verification techniques like model checking, assertion based verification and symbolic analysis to fault injection are discussed below.

Scott et al [44] presents a methodology called ABVFI which is fault injection based on assertion based verification. ABV is a formal verification method that checks for critical properties at the hardware level by embedding assertion statements into the hardware design (RTL). ABVFI proves selected properties of the system in the presence of faults. The fault space for this model checking method, considers an exhaustive state and fault space. Krautz et al [45] used formal verification to exhaustively measure the fault coverage of a VHDL design. Symbolic simulation of sequential circuits are performed by creating Binary Decision Diagrams of the circuit's state space. The BDD analysis helps in identifying the fault detection/correction capability of each state in the system and quantitatively classify them, after a fault injection. This fault injection module (FIM) exhaustively covers only the state space and not the input data space. The input data and faults are fed into the FIM as bit vectors. Leveugle [46] introduced the new approach of combining property with mutation of the circuits to conduct fault injection experiments on circuits. Mutants of VHDL circuits that exhibit faulty behavior similar to the ones expected in the field

are prepared through a controlled generation approach. Critical properties are checked for any violations in the mutated circuits thereby helping in identifying the undesirable effects of multiple faults in the circuit. Several commercial tools for property checking, in particular Verifier from Safelogic and FormalCheck from Cadence are used for his experiments. Daniel et al [47] was the first one to demonstrate the effectiveness of symbolic fault injection on an SIHFT application in Java platform. This simulation based fault injection approach involves the injection of symbolic faults during the symbolic execution of the software. Transient bit flips in the data memory variables are simulated by executing pseudo statements in the code during symbolic execution of the code. The critical properties of the software are proven for validity for all inputs and all modeled faults using formal software verification tools like KeY. All the above referenced works have combined fault injection on hardware design with formal verification. Whereas, this thesis focuses on applying the fault injection based on formal verification principles at the functional model level.

There are few works in the literature on model based fault injection. MODIFI [48] is a model implemented GUI based fault injection tool which gives a good level of fault injection controllability and observability on Simulink behavior models. Fault models are derived from XML descriptions and implemented as Matlab code in the saboteurs inserted in the fault location during runtime. The evaluation of the FI experiments are carried out either by using runtime assertions or by comparison with nominal behavior. MODIFI additionally facilitates minimal cut set analysis to find the minimal set of faults that violate a safety requirement. Moradi et al in [49] presents another automated fault injection framework in Simulink model based environment. The model is annotated by the designer with custom FI blocks that can be configured with the type of fault, fault injection time, duration and other parameters. These annotations are transformed into saboteurs and then the FI experiments are run one by one on the FI deployed model using an FI orchestrator script. In addition to hardware faults, the authors of this work also consider software faults that include code insertion and code drop as well as latency which are not considered in [48]. Pill et al in [50] proposed a toolset SIMULTATE that combines fault injection and mutation testing on Simulink behavioral models. The fault library mainly consists of structural mutations to blocks in the model, that include adding new block, replacing blocks and rerouting signal connections. The major advantage of this toolset is its flexibility to easily support custom fault models. In all of these model-based fault injection frameworks, though the saboteur insertion is automated, the placement of saboteurs is decided by the modeler and therefore the FI tests could end up in insufficient fault coverage problems. FI experiments conducted in a classical FI approach, with the need to feed in input vectors for executing the model, suffers from incomplete input, state and fault space coverage.

Chapter 4 Overview of the Representative System

This short chapter gives a brief overview of the representative safety critical system on which the model-based design assurance workflow and the property based fault injection are applied. The target system in this study is a FPGA-based digital I&C architecture for safety critical applications in nuclear power plants.

SymPLe architecture [51] was created to solve the risk of software common cause failures when transitioning from analog to digital technology in safety-related instrumentation and control applications in nuclear power plants. Specifically, SymPLe is a notational architecture designed in a MBE environment which aims to ease verification of digital systems using complexity-aware design and design-for-verification principles. SymPLe is a portable FPGA overlay architecture for PLC computation. SymPLe is programmed with a Function Block language similar to IEC-1131 PLC programming languages. The adoption of PLC's function block programming notation in SymPLe was due to the long history of PLCs in safety and non-safety operations in nuclear power. The constrained and complexity-aware SymPLe design supports deterministic execution of elementary functions, known as Function Blocks. This helps in porting the SymPLe architecture into existing PLC applications which are common in nuclear power and other process control applications. Figure 4 shows an overview of the SymPLe architecture.

The design of SymPLe is implemented in three basic control hierarchies: the global sequencer, local sequencers or tasks, and a complete set of Function Blocks (FB) per local sequencer/task. In SymPLe, each task lane execution happens independent of the other tasks. The global sequencer (GS) is the highest controlling component of the architecture. The global sequencer is responsible for coordination of each task and the marshaling of data to/from task registers from/to IO points in the architecture. Triggering task lane execution is the responsibility of the scheduler contained in the GS. During the initialization of the architecture, the GS reads from the global sequence and sets the task schedule or hard deadline for each task, initializes task memory locations, and initializes outputs of the system. Following this, all system inputs are read into the system and task lanes are executed. When a non-recoverable error is encountered in the architecture, the GS transitions to the failed state in which the architecture terminates task executions and all outputs are put in a safe state. SymPLe architecture implements fail-stop and fail-safe behavior in this manner.

Each task contains a local sequencer, a function block controller, a set of function blocks, and a redundant memory. The local sequencer is the controller for the task lane. It reads the task sequence programmed by the user and schedules the execution of the function blocks as per the user-provided deterministic sequence to implement the application in the task. The local sequencers' function is to marshal data to/from task registers and from/to function block registers and to sequentially trigger function blocks in its own task lane. The local sequencer is implemented using state flow diagrams with several main and sub states. The main states within Local sequencer state machine are INIT, FB_SELECT, FETCH, FB_EXECUTE, WRITE, FB_DONE, TASK_DONE and ERROR. If an error detected during function block execution is identified as a recoverable error, the local sequencer retries by restarting the FB execution. If the error is identified as non-recoverable, Local Sequencer informs Global Sequencer and the GS triggers an abort of the current task execution and restarts the entire task sequence. The FB Controller supervises and controls the execution process of the selected function block. It is implemented as a state machine with states mirroring those of the local sequencer. Functions blocks receive data into shift registers, process the data as per intended functionality, and computes and writes results in their output registers.

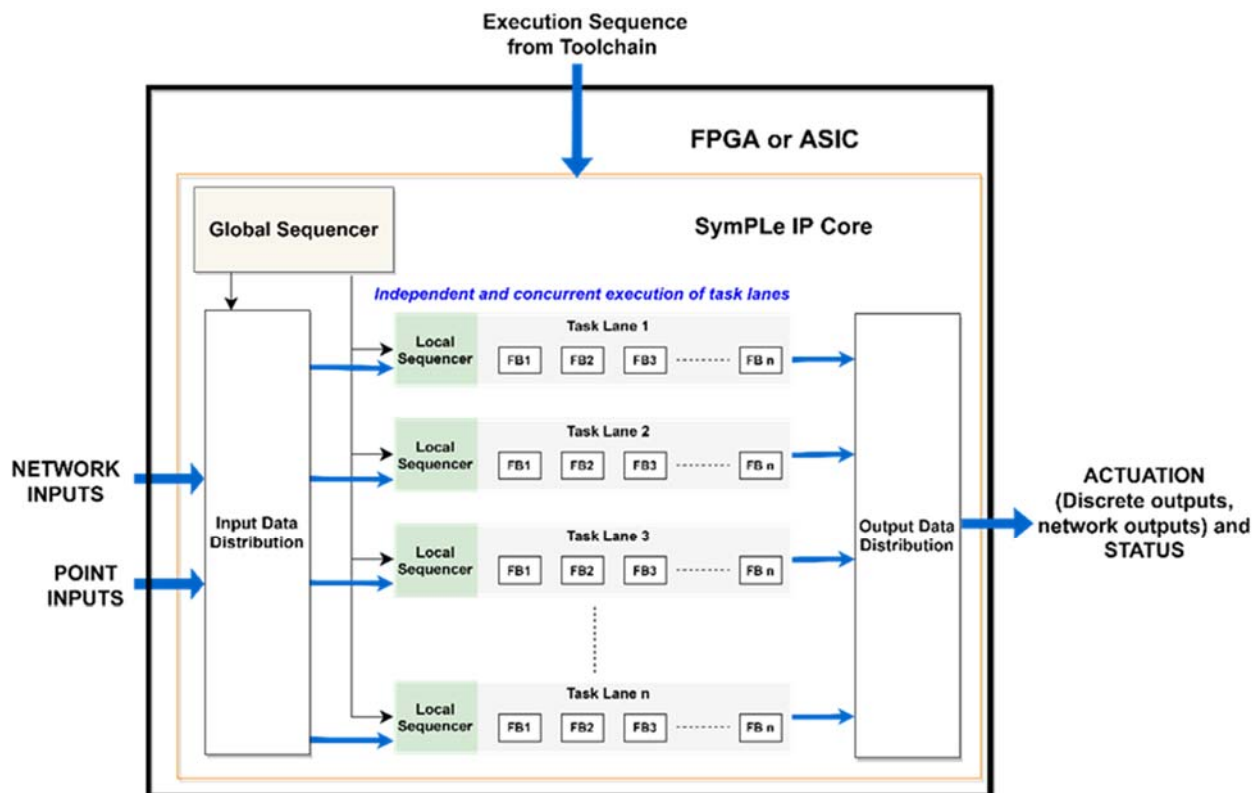


Figure 4: The SymPLe Architecture [42]

Function Blocks are the elementary “program or computation” units that SymPLe architecture employs for its program organization or application building. The function block model is derived from two IEC standards – IEC 61131-3, and IEC 61499. The 32 different function blocks implemented in SymPLe architecture to aid in application building are given in Table 1. As shown in Figure 5, the fault detection and tolerance mechanisms in SymPLe architecture starts at the lowest level of architecture which is the function blocks thereby allowing the system to enter a fail-stop state before the error propagates beyond boundaries of the system.

Table 1: Implemented Function Blocks in SymPLe [51]

| Operation Type | Functionality | Valid Data Type(s) |
|-----------------|--|--------------------|
| Binary | NOR,AND,NOT,OR,XOR,NAND | BOOL,SAFEBOOL |
| Bitwise | BNOR,BAND,BNOT,BOR,BXOR,BNAND,BSL,BSR | INT |
| Comparison | MAX,MIN,MUX,GT,GE,EQ,LT,LE,NE | INT,QMN |
| Arithmetic | ADD,SUB,MUL,DIV | INT,QMN |
| System | MOV | Any |
| Type Conversion | QMNtoINT, INTtoQMN, BOOLtoSAFEBOOL, SAFEBOOLtoBOOL | Various |

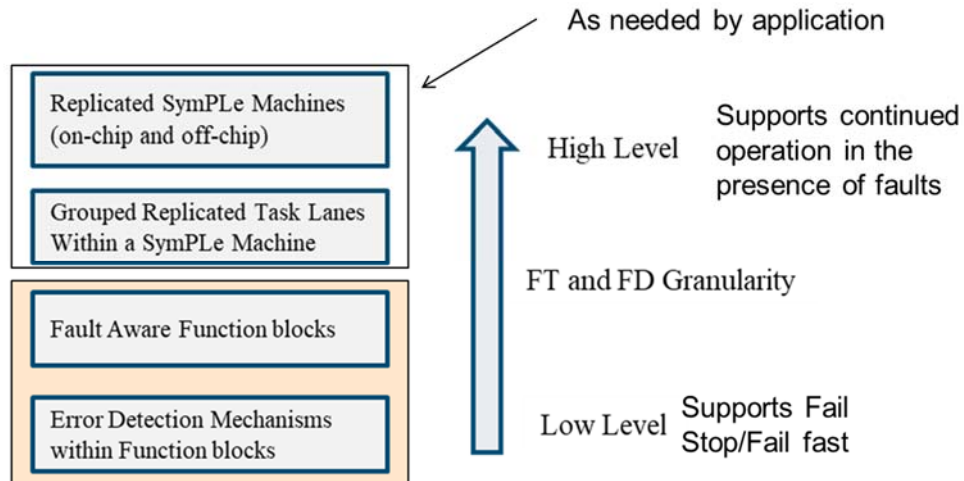


Figure 5: Fault Tolerance Approach for SymPLe [51]

Chapter 5 Model Based Engineering and Design Assurance

This chapter gives an overview on Model based Engineering and Design and the general model based Design assurance workflow for safety critical systems, starting from textual requirements to the hardware implementation, from a Mathworks perspective. The different choices of MBD tools available and the challenges faced in the development of the end to end design assurance workflow for the representative architecture are also presented in this chapter. This chapter then presents an overview of the Model based Design Assurance workflow developed for SymPLe architecture, and then a more detailed discourse on the different stages of Model testing process that involves, unit, integration and system testing. The chapter also gives key examples of test failures caught in the process of model testing and highlight the capabilities of model simulation and testing in finding a number of critical design flaws in the model.

5.1 Model-based Engineering and Design

Model Based Design and Engineering is not a new practice, but has evolved over years to be a discipline in itself – merging various scientific computing and engineering disciplines together to provide a comprehensive approach to engineering and integrating complex systems. We note the phrase “Model Based Design” has been used in many different contexts to the point where it’s meaning and purpose is somewhat unclear. For safety critical systems design, we use a more refined definition taken from the systems engineering and the formal methods community.

“Model Based Design Engineering (MBDE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle”[52]

The increasing use of *embedded software* in domains such as automotive, railway, aerospace and industrial automation has resulted in very complex systems that have been proven difficult to manage with conventional HW-SW co-design approaches. Due to its capability to address system integration, complexity, co-design and productivity challenges, MBDE is quickly becoming the preferred engineering paradigm for a number of industries. In MBDE, a *model* is introduced into the system lifecycle between requirements/specification and the code. The essence of MBDE workflow starts with an initial conceptual graphical model representing the software (or hardware) component under development usually developed by the system engineers. While conceptual system models are abstract models capturing the

requirements, design decisions, system interactions and structural aspect of the system, they do not contain details of the system SW or HW implementation. These conceptual models serve as the base for the development of the more detailed executable models by the SW developers. These executable models represent the dynamic behavior aspect of the system and can be executed to extract the temporal response of the model when stimulated with inputs. Executable models help to represent discrete-time, continuous-time and discrete-event domain interactions between the controller and plant environment in embedded applications and cyber physical systems. The executable models undergo many refinement iterations until it becomes sufficiently detailed to serve as the blueprint for the final implementation through automatic code generation.

Model based engineering has been used widely in automotive, avionics, railway, healthcare and other safety sectors due to the several benefits that it offers:

- Reduction in cost and time for development
- Higher product quality.
- Standardized medium for sharing artifacts between stakeholders that include the customers, system engineers, software and verification engineers
- Improved complexity management of systems.

The early realization of the system design through executable models helps gain more insight on the design shortcomings and flaws which are usually overlooked in the requirements and specification phases in traditional Software Development Life cycle (SDLC). The most significant aspect of productivity gains from modeling in the context of embedded software design is the capability to automatically generate high level source code (C, VHDL, Verilog etc), from the behavioral model representations. Model based engineering also greatly supports the system verification and validation activities by providing integrated tools for conducting model testing, debugging, formal and static verification. The formalism inherent in the modeling language offers a consistent and coherent central platform for specifying the requirements, design and verification which is very essential for safety-critical system design.

Unlike the traditional SDLC process with all phases being clearly separated from each other, Model based Design and Engineering offers a single framework with continuous integration of the SDLC phases. MBDE tools like MathWorks Simulink® or Ansys SCADE®, provides a central platform for representing executable graphical models with qualifiable code generators. In addition to providing a variety of libraries with pre-verified blocks to model the functional behavior of hardware or software

systems, these tools provide many toolboxes for conducting analysis and verification activities from functional to code level. Some of these toolboxes include the static analysis, design error detection, property proving, model coverage, and testing toolboxes. This thesis studies the application of model based engineering and design assurance in an IEC 61508 context to an I&C architecture for nuclear industry and collecting results and experiential findings that can inform the safety critical community on the benefits and pragmatic aspects of MBDE&V. Several modeling practitioners has claimed that MBDE helps in finding errors very early in the developmental cycle thus leading to less iterations per cycle and increasing productivity gains of up to 4 to 10 times [53]. But there has been very few studies and evidences available in the open literature on the applicability of these claims to the Nuclear industry.

5.2 Model Based Design Assurance

Figure 6 below gives the general verification workflow in model-based engineering. The design process starts from textual requirements to an executable specification, which is the initial model representation of the system, that can be executed. As this executable specification gets more and more refined and matured enough for production code generation, the verification process starts off. Module level functional testing that verifies if the model implements all the requirements correctly, is performed. Following the unit and integration testing on models, reviews and static analysis are performed on models to find design issues and validate the adherence of models to modeling guidelines and safety standards. Thus, model/design verification phase helps in discovering design errors at design time. Once a good level of confidence is achieved on the models, the next step is to validate the generated code against the model. The model and code are stimulated with identical test vectors. Verifying the equivalence in the test results from model and code ensures that the generated code represents the model semantics correctly without any translation flaws during automatic code generation [54]. Running tests on code also ensures that the code behaves as expected without any unintended functionality. After the generated HDL code is verified, the HDL is synthesized, bitstream is generated, placed and routed on the FPGA. Hardware testing and verification is performed to ensure that the system behavior is consistent, when implemented on actual hardware. Thus, code verification phase helps in gaining confidence on the generated code and hardware verification phase helps in gaining confidence on the hardware (FPGA) implementation of the system.

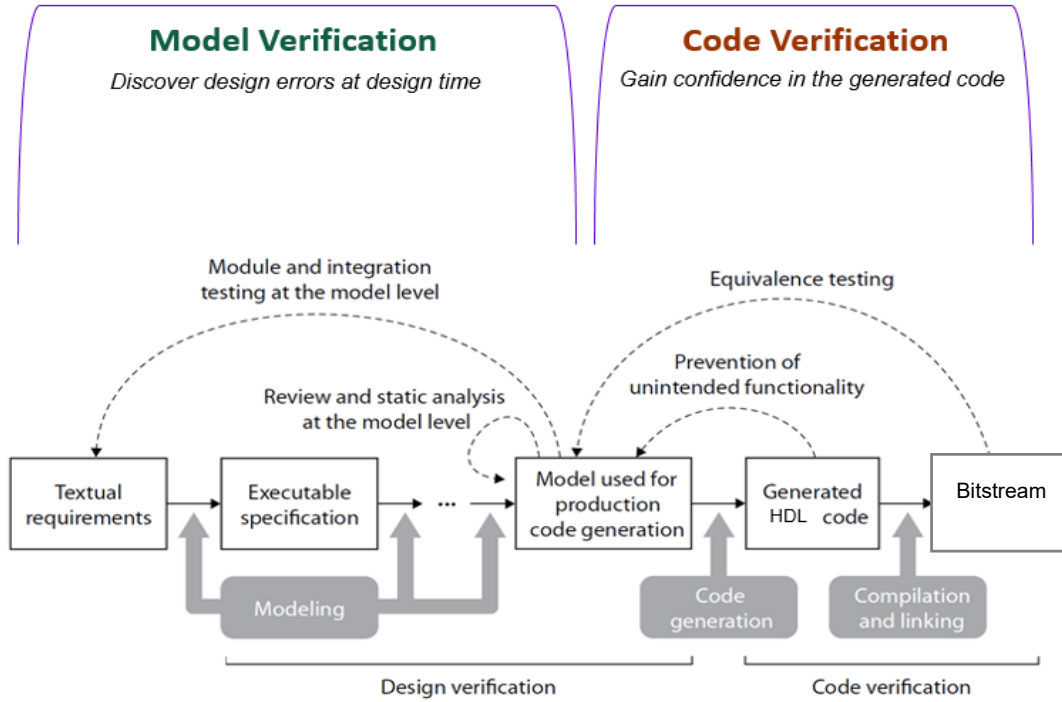


Figure 6: Model-based Design Verification Workflow [54]

5.3 Choice of MBD Tool

Several MBD tools to be used for real time systems were assessed during the preliminary phase of this research effort that includes SCAD Suite from ANSYS [55], MATLAB Simulink® and IBM Rhapsody. These tools also offer a strong mathematical basis for formal verification. Additionally, these tools are tailored for safety critical applications by being certified under IEC 61508 at SIL 3.

Simulink® tools were chosen for the design and V&V of the SymPLe I&C architecture as they offered an end-to-end solution under a central modeling platform for requirements, design, simulation, testing, formal verification, static and coverage analysis for all embedded systems including the FPGA based systems. Simulink HDL Coder generates portable and optimized HDL code and Simulink HDL Verifier™ helps test and verify the Verilog® and VHDL® designs for FPGAs. Simulink development and verification toolchain helps to preserve traceability from requirements to bit-stream. Simulink provides a formal verification tool from Prover technologies called the Simulink® Design Verifier (DV). This has been extensively used in verifying the functionality of the representative system and to

exhaustively perform fault injection. The formal verification is also extended to the lower levels of HDL code synthesizable on FPGA or ASIC using Questa PropCheck from Mentor Graphics [56].

The tool boxes in Simulink used in verification of SymPLe are given in below Table 2. The first column shows the products for use in Mathworks that comply with the IEC 61508 standard. The major tools used in the SymPLe design verification process that includes Simulink Check, Simulink Test, Simulink Coverage and Simulink Design Verifier are all IEC 61508 certified.

Table 2: Table showing IEC 61508 certified Mathworks Products

| | IEC 61508: 2010 (SIL 1-3) | ISO 26262: 2011 (ASIL A-D) | EN 50128: 2010 (SIL 1-4) | IEC 62304: 2006 (SIL 1-3) | IEC 61511: 2003 (SIL 1-3) |
|--------------------------|---------------------------------|----------------------------------|--------------------------------|---------------------------------|---------------------------------|
| Embedded Coder | ✓ | ✓ | ✓ | ✓ | |
| Simulink Check | ✓ | ✓ | ✓ | ✓ | |
| Simulink Coverage | ✓ | ✓ | ✓ | ✓ | |
| Simulink Test | ✓ | ✓ | ✓ | ✓ | |
| Simulink Design Verifier | ✓ | ✓ | ✓ | ✓ | |
| Polyspace Bug Finder | ✓ | ✓ | ✓ | ✓ | |
| Polyspace Code Prover | ✓ | ✓ | ✓ | ✓ | |
| Simulink PLC Coder | ✓ | | ✓ | ✓ | ✓ |

5.4 Development of a model based design assurance workflow

One of the unanswered problems for safety solutions engineers is the question of how to address the difference between safety guidelines codified in standards and the existing MBDE practices and tool capabilities. Standards like IEC 61508 include directions, recommendations and tables on appropriate procedures and strategies available for observance, during design and verification of safety critical systems. However, the standards do not include guidance about the workflow to be followed in a MBDE context. MBD Tools like Mathworks Simulink® provides certified tools, but the practitioner is required to "build" the workflow to meet the safety standard.

The detailed design assurance workflow of SymPLe architecture given in Figure 7 is developed by carefully reviewing the IEC 61508-3 standard and identifying all the recommended verification practices for SIL levels 3/4. Table 3 gives a one to one mapping of the V&V workflow steps and the corresponding IEC 61508 requirement that it satisfies. This workflow is derived from the generic model based design assurance workflow in Figure 6. The end-to-end design verification of the SymPLe architecture spans from verification of model to verification of generated HDL code and finally the hardware implementation. Each step in the verification process is premeditated to satisfy the criteria for IEC 61508 compliance and ensures that the design artifacts aptly reflects the requirements and accurately depicts system behavior while still maintaining bi-directional traceability between all design and V&V artifacts. In this manner, each step in the verification workflow helps build evidence for IEC 61508 assurance.

In the presented design assurance workflow in Figure 7, the black arrows indicate the design and development workflow. The development starts with Requirements Elicitation followed by the design of the architectural and behavioral model, then the RTL Code Generation and finally programming the Target FPGA with the HDL Code Bitstream. The blue arrows indicate the verification workflow with verification artifacts being listed out for each of the verification cycle. The green boxes indicate the development/verification artifacts that has traceability links to other development artifacts and green arrows indicate the traceability links between the various levels of development.

The Traceability links (1) indicate the relationships between high level requirements and mid/low level requirements. High level requirements specify the general properties that is applicable to the entire system architecture and mid/low level requirements give detailed specifications regarding the implementation of each individual components in the architecture and their interactions. Mathworks Simulink environment allows for quick navigation between the model and requirements using traceability hyperlinks. Traceability links (2) helps in relating requirements to corresponding model objects. Each block, state or state transition or in general any model object can be linked to the requirement which it is implementing. Traceability links (3) provides links between the auto-generated code and the model objects and also allows for quick navigation from the code to the model objects. Traceability links (4) are links between the requirements and HDL Code. This maps the snippet of HDL code to the requirement it implements and provides quick navigation between the Requirements Manager and the HDL Code in the Simulink Environment.

Table 3: V&V workflow phases to IEC 61508 Requirements

| V &V Artifacts and Bidirectional Traceability | IEC 61508 Part 3 and IEC 61508 Part 7 |
|---|--|
| Bidirectional Traceability between Requirements and Model | Annex A Table A.2 – Software design and development – software architecture design <ul style="list-style-type: none"> Forward traceability between the software safety requirements specification and software architecture Backward traceability between the software safety requirements specification and software architecture |
| Bidirectional Traceability between Requirements, Test Cases and Formal proofs | Annex A Table A.9 – Software verification <ul style="list-style-type: none"> Forward traceability between the software design specification and the software verification (including data verification) plan Backward traceability between the software verification (including data verification) plan and the software design specification |
| Bidirectional Traceability between Requirements, Model and Code | Annex A Table C.6 – Properties for systematic safety integrity – Programmable electronics integration (hardware and software) <ul style="list-style-type: none"> Forward traceability between the system and software design requirements for hardware/software integration and the hardware/software integration test specifications Backward traceability between the software safety validation plan and the software safety requirements Specification |
| Model Based Testing and Coverage Analysis | Annex A Table A.5 – Software design and development software module testing and integration <ul style="list-style-type: none"> Model Based Testing Dynamic Analysis and Testing Annex B Table B.2 – Dynamic analysis and testing <ul style="list-style-type: none"> Structural test coverage (entry points, branches, statements)100 % Structural test coverage (conditions, MC/DC) 100% Test case execution from model-based test case generation |
| Formal Verification and property proving | Annex A Table A.5 – Software design and development software module testing and integration <ul style="list-style-type: none"> Formal verification Annex A Table A.9 – Software verification <ul style="list-style-type: none"> Formal Proof |
| Static Analysis | Annex A Table A.9 – Software verification <ul style="list-style-type: none"> Static Analysis Annex B Table B.8 – Static analysis <ul style="list-style-type: none"> Static analysis of run time error behavior Boundary Value Analysis |
| HDL Testing and Coverage Analysis | Annex E E.6 Functional test on module level Annex E E.7 Functional test on top level Annex E E.13 <ul style="list-style-type: none"> Coverage of the verification scenarios (test benches) Dynamic Analysis and Testing Annex A Table A.2 <ul style="list-style-type: none"> Automatic software generation |
| Equivalence Testing between Model and HDL Code | Annex E E.5 <ul style="list-style-type: none"> HDL simulation Annex B Table B.2 – Dynamic analysis and testing <ul style="list-style-type: none"> Test case execution from model-based test case generation |
| Formal Verification and property proving: Code | Annex A Table A.5 – Software design and development - software module testing and integration <ul style="list-style-type: none"> Formal verification |
| FPGA in Loop Testing | Annex B Table B.2 – Dynamic analysis and testing <ul style="list-style-type: none"> Test case execution from model-based test case generation |
| Runtime Monitors | Annex A Table A.2 3b – Software design and development – software architecture design <ul style="list-style-type: none"> Diverse monitor techniques (with independence between the monitor and the monitored function in the same computer) |

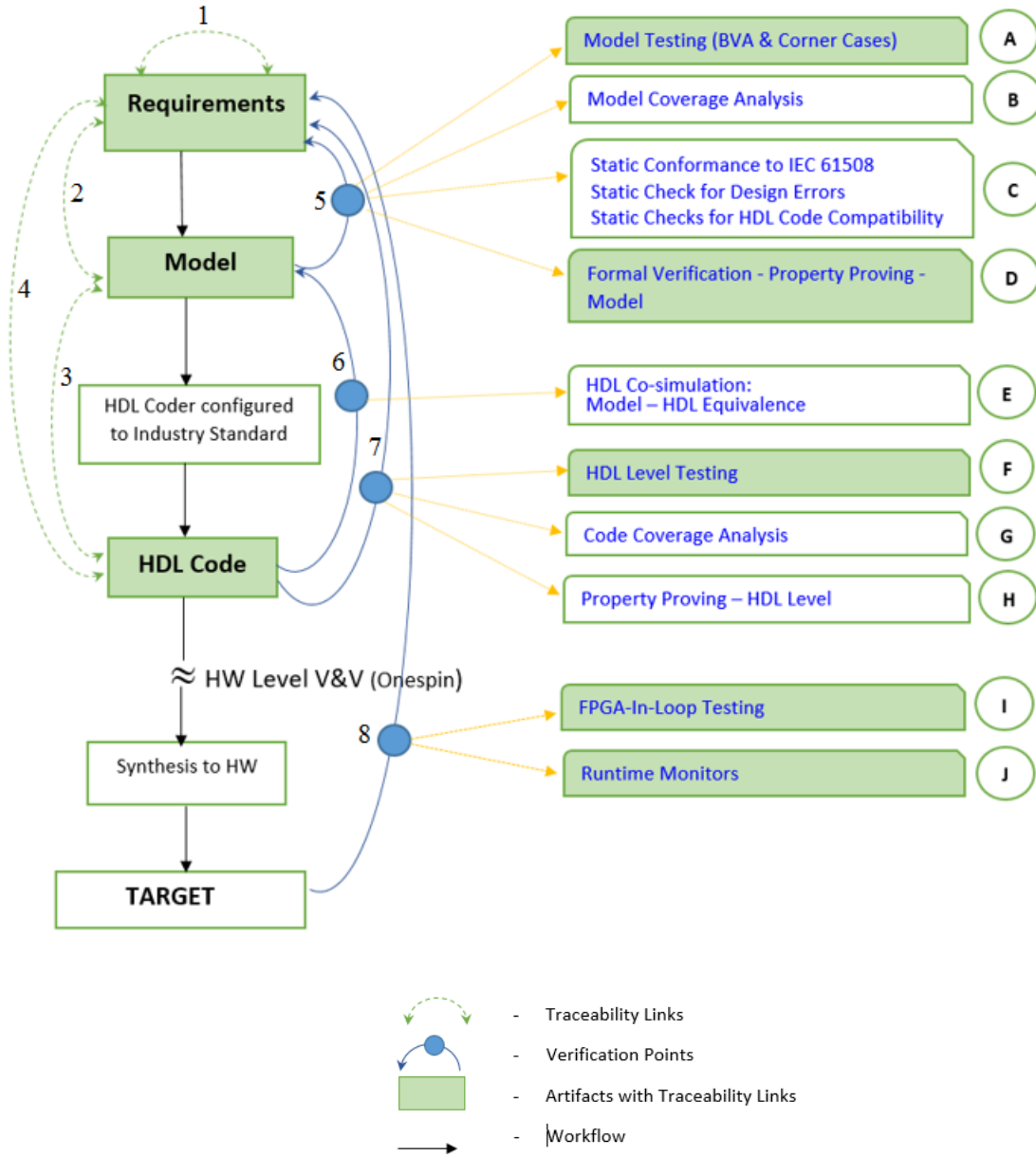


Figure 7: V&V Workflow for SymPLe architecture

A brief overview of the steps in the verification workflow is provided in this section.

The first verification phase (5) verifies the model against the specified requirements. Following are the different verification techniques applied on the first development artifact, the model.

(A) Model Testing (MT) – Model Testing involves dynamically verifying the model against the specified requirements. Boundary value Analysis and Corner cases are considered for coming up with error sensitive testcases. Bi-directional traceability links help in connecting the testcases to the corresponding requirements. Simulink environment facilitates easily navigating from requirements to the testcases and vice versa. This step involves higher manual effort compared to following steps that are fully automated or partially automated. Issues in testcases, design and even requirements are caught here. The process tends to be a cyclical process that involves test execution, test failure analysis, testcases/requirements/design update and again test execution and this process continues until the testcases, requirements and design are stabilized/finalized. This step forms the base for the following automated/semi-automated verification steps, as this is where we develop a good understanding of the component under test. This step is described in detail in below sections in this chapter.

(B) Model Coverage Analysis – This process involves measuring the test coverage on the model and adding additional testcases to cover the model execution paths that were not covered with original test vectors. The model testing and structural coverage analysis is an iterative process and together helps build a complete set of test vectors. The coverage analysis step is described in detail in Chapter 6.

(C) Static Verification – This process involves statically analyzing the model to find design errors and any violations to the IEC 61508 safety modelling guidelines. This step is described in detail in Chapter 7.

(D) Property Proving at Model – This is formal verification performed at the model level. Although testing can expose many design flaws, it is impossible to test a system exhaustively. Formal verification methods have a sound mathematical basis and are defined by a formal specification language. Model-based property proving is performed using the Mathworks Simulink Design verifier. Property proving involves proving safety critical properties that shall always hold true in the model component. Simulink DV checks the entire state space and proves or disproves the critical property. If a property fails, a counterexample is generated showing the input sequences and state conditions that resulted in the property failure. Bi-directional traceability is also ensured between the properties and requirements and Simulink environment allows in navigating from requirements to the modeled properties and vice versa.

(E) HDL Co-simulation - The second verification phase is (6) and involves verifying the code against the model. HDL/Register Transfer Level code describes the flow of signals in the digital circuit using logical operations and programming constructs. Simulink is used in conjunction with ModelSim HDL Simulator to verify the equivalence between the Simulink model and the HDL code behavior. Using HDL Co-simulation, a communication link is created between Simulink and the HDL simulator. The equivalence between Simulink model and the HDL code is proven by concurrently executing both the model and code with the same test vectors and confirming the equality of the output behavior. Any difference in output behavior could indicate a mismatch in the dynamic behavior of the code and model.

The third verification phase is (7) and involves verifying the code against the specified requirements. Following are the verification techniques at this verification phase.

(F) HDL Code Testing - This process uses the Software-in-Loop testing in Simulink to run the test vectors directly on the HDL Code and verifies if the HDL Code complies with the requirements.

(G) Code Coverage Analysis – During HDL code testing, test coverage on the code is collected from ModelSim HDL Simulator. Code coverage helps to assess if the tests are good enough to completely cover all lines and other aspects of the code. The several coverage metrics applicable for HDL code are Statements, Branches, Conditions, Expressions, Toggle, Finite State Machine (FSM) States and FSM Transitions. This test coverage information is further used to update the test vectors to cover the uncovered parts of the HDL code and improve the quality of the code level tests.

(H) Property Proving at HDL Level – Safety critical properties of the system having being proven at the model level using Simulink Design verifier, needs to be proven for validness at the code and hardware level. Mentor Graphics Questasim tool is used to formally verify critical properties of the design at the HDL code level. Questasim uses Assertion Based Verification (ABV) to verify the HDL code. ABV offers a powerful formal verification paradigm for hardware by exhaustively verifying critical properties which are important for the correct functionality of the system. The properties are embedded into the HDL code as assertions and an error is flagged if the assertions fails to hold true. The two most widely used ABV languages to specify the property for proving are Property Specification Language (PSL) and System Verilog (SVA).

GateLevel/NetList – The netlist that is generated during synthesis of the HDL code is the gate level representation of the design and describes the functionality using digital gates and how they are interconnected. This netlist level verification is supported by low level verification tools like One Spin. Equivalence checking and formal verification at this level is planned as future work. The last verification phase (8) involves verifying the hardware implementation of SymPLe against the specified requirements.

(I) FPGA-In-Loop Testing - After placing, routing and programming the SymPLe bitstream onto the FPGA, FPGA-In-Loop Testing is performed that runs the Test vectors directly on the programmed FPGA. The HDL Verifier tool in Simulink is used to test the HDL code implementation on an actual FPGA using FPGA-in-the loop simulation environment. It provides connection between the FPGA board and the test simulations in Simulink.

(J) Runtime Monitors – All the previous steps of verification are conducted at the design time. Though the SW design could be now free from design flaws, it cannot be guaranteed that the system would operate flawlessly and safely as it is still prone to physical faults and cyber-attacks during runtime. There could also be residual design flaws that might have bypassed the testing phase. To ensure that critical properties of the system are always held true even during runtime, external monitors that observe the execution behavior of the target system during runtime are employed. During runtime verification, the system functionality, communication and interactions are observed while applying any constraints dynamically. We used a tool known as MBAC [57], developed by the researchers at McGill University, Canada, to generate runtime assertion-based monitors. MBAC generates Verilog automata checkers from the properties written in the Property Specification Language (PSL) or System Verilog properties temporal languages. The Verilog runtime monitor code is implemented on an FPGA along with the SymPLe application that is to be monitored.

Detailed description of the processes followed and results obtained during model testing (A), model coverage analysis (B) and static verification (C) are provided later in this and subsequent chapters. More details on the results obtained for the steps (D), (E), (F), (G) (H), (I) and (J) can be found in [51].

5.5 Model Testing

As SymPLe architecture is a fully model-based architecture, the first design artifact to be verified is the ‘model’ itself. Similar to Model-based design, Model-based verification also takes advantage of the simulation capabilities of the executable model in the modelling environment. Model verification enables

finding design errors at design time itself, very early in the development cycle, thereby drastically reducing rework effort. Model testing is the first step in the verification workflow and is essentially the dynamic verification process on the models. It involves preparing test vectors and feeding in inputs to executable models, executing the models and evaluating the model outputs. Model testing is conducted hierarchically in three stages: Unit Testing, Integration and System/Application testing. It is described in detail in the next section. Model Testing applies effective techniques such as Boundary value analysis, Equivalence Partitioning and Corner case testing. These methods have been very useful in developing powerful and optimized test sets that helped reveal many difficult to find bugs present in the design.

Boundary Value Analysis and Equivalence Partitioning: Boundary value analysis attempts to discover program errors for input values at the boundary of equivalence classes. Values at the borders of a range of values are better suited to form error sensitive test cases than values in the middle. Boundary Value Analysis and equivalence space partitioning are the most common methods used to reduce the vast input domain of a system under test, to a tractable set of values. Equivalence partitioning involves splitting each variable's domain into parts such that all values in one part display the same logical behavior within the program. For a variable whose range is n to $+n$, an exhaustive test set is expected to cover the full span of $2n+1$ values. This large input space is restricted by analyzing the application of variables in branching and decision predicates within the code. For variables that are part of a predicate clause or guard expression a sample set of values satisfying the condition and on both sides of the condition are sufficient to test all necessary interactions.

Corner case Testing: Corner case test cases involve setting multiple input parameters at their maximum or minimum values simultaneously or testing the component outside of the base assumptions. If developers have overlooked the corner cases during their design and development of the software, it could cause the system to behave incorrectly or unexpectedly when inputs take extreme/unexpected values. A dependable system is expected to always remain in safe or defined state even with unexpected or invalid input conditions. While testing the SymPLe architecture, especially the computationally rich components like function blocks, we have considered invalid input values and input combinations that do not normally appear during regular operation. Components with more of state machine logic have been tested with edge case sequences, which involve unexpected/rare sequences of inputs to the state machine.

5.5.1 Hierarchy of Model Testing

Model Testing involves formulating test vectors from the functional requirements for the model, execution of the test cases on the model and assessing and verifying the model behavior [27]. Our test method follows the principles of White Box Testing. Similar to how traditional software testing progresses, model testing follows a hierarchical approach. Figure 8 shows the different hierarchy levels of model testing starting with unit testing, then integration testing and finally system testing. In unit testing, each component in the SymPLe architecture or subsystems within complex model components eg: individual function blocks (MUX, ADD, DIV.. etc), subsystems within function block controller, local sequencer and global sequencer components needs to be tested independently to ensure that they adhere to their detailed low level requirements. In unit testing the inputs coming into the component from other components becomes the test input parameters and signals going outside of the test component becomes the test outputs to be assessed. 100% functional coverage and >95% structural coverage are aimed at, during the white box unit testing phase.

Integration testing phase ensures that two or more units when integrated together interact with each other and behave as defined in the requirements. It checks if the interfaces between two or more components have the same data type and signal ranges and the timing behavior of internal signals shared between model components are consistent in order to ensure proper interaction between components thereby achieving correct system operation. Interactions between Function block controller and function blocks, Local sequencer and Function block controller, Local Sequencer and Global Sequencer etc. are to be considered for testing in this level. Testing the behavior of each task lanes can be considered to be part of the integration testing phase. The next level of testing is system testing where the fully integrated system functionality is tested as a black box. In the context of SymPLe architecture, system testing ensures that all task lanes operate well, interacts properly with the global sequencer, thereby meeting the purpose of the system. Testing an application built on top of SymPLe architecture is the methodology adopted for System testing of this I&C architecture.

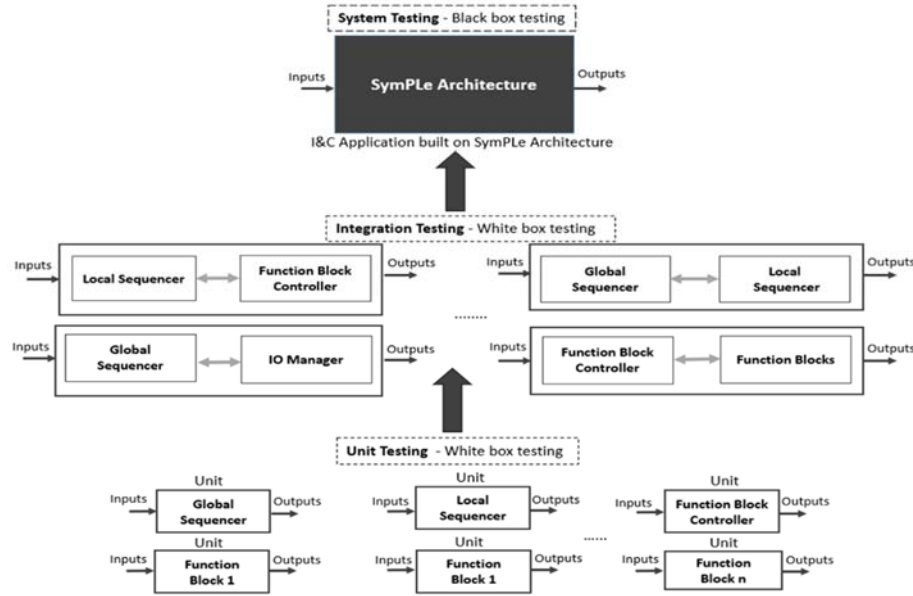


Figure 8: Model Testing Hierarchy as applied to SymPLe architecture

5.5.2 Simulink Tool Major Features to assist model testing

Simulink Test is toolbox integrated with Simulink that has several features to support model-based testing. It provides user friendly interfaces to write test sequences, manage, and execute systematic, simulation-based tests on models, generated code, and simulated or physical hardware.

There are three major features for Simulink Test:

Test Manager: Test Manager (Figure 9) is the interface where we create and manage testcases, execute them and review test result summary. Testcases can be grouped into different Test suites and can be executed individually or as a batch in the Test Manager. Test manager also has features that eases analyzing of Test results. As shown in Figure 10, for each testcase, Test Manager provides a graphical interface to visualize the behavior of signals in the model and the verify steps in the testcase with respect to time, during the testcase execution. We can also link the testcases to Requirements from the Test manager to enable identifying the requirement that a particular test case is verifying. Additionally, we can export test reports with all of the following details, test requirements, test descriptions, test steps, coverage results, plots for criteria, assessments and simulation outputs, error and log messages from the Test manager.

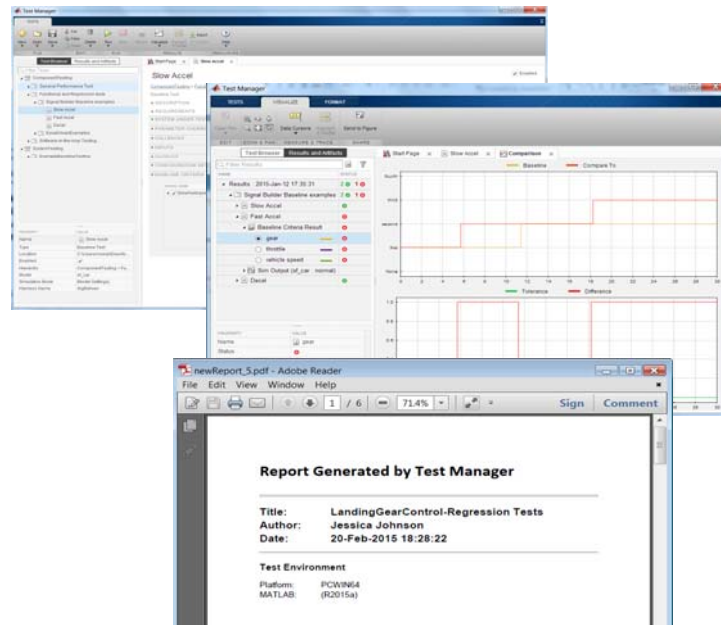


Figure 9 : Test Manager Interface [54]

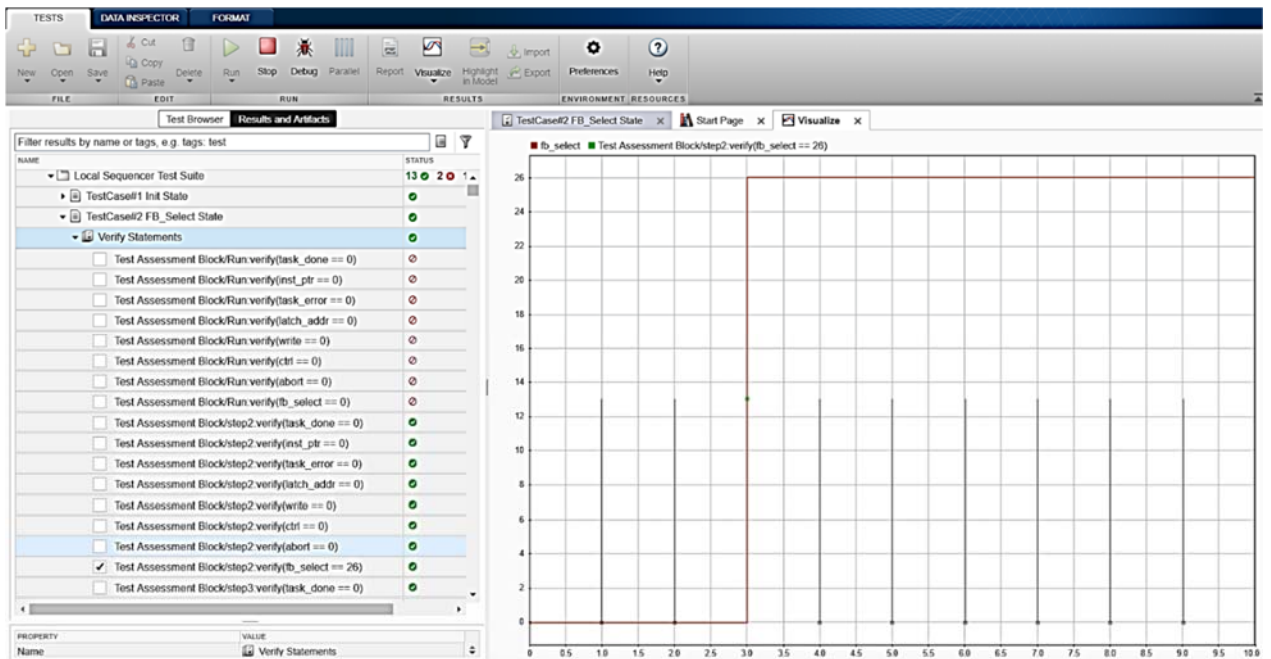


Figure 10: Graphical representation of test assessment results and design signals in Test Manager

Test Harness: Test Harness (Figure 11) is a synchronized simulation test environment wherein the component under test is isolated for testing. Test harness can be created at unit (subsystem) level or system level. It also provides a synchronized test environment (harness to model). Any changes made in the main model gets automatically reflected in the Test Harness model. The Test Harness can be configured for Model-In-Loop, Software-In-Loop and FPGA-In-Loop testing. Test harness can be created for just one subsystem or a part of the main model that needs to be tested independently during unit testing. Simulink automatically identifies and assigns ports for the inputs and outputs of the subsystem that is contained within the test harness.

Test Harness

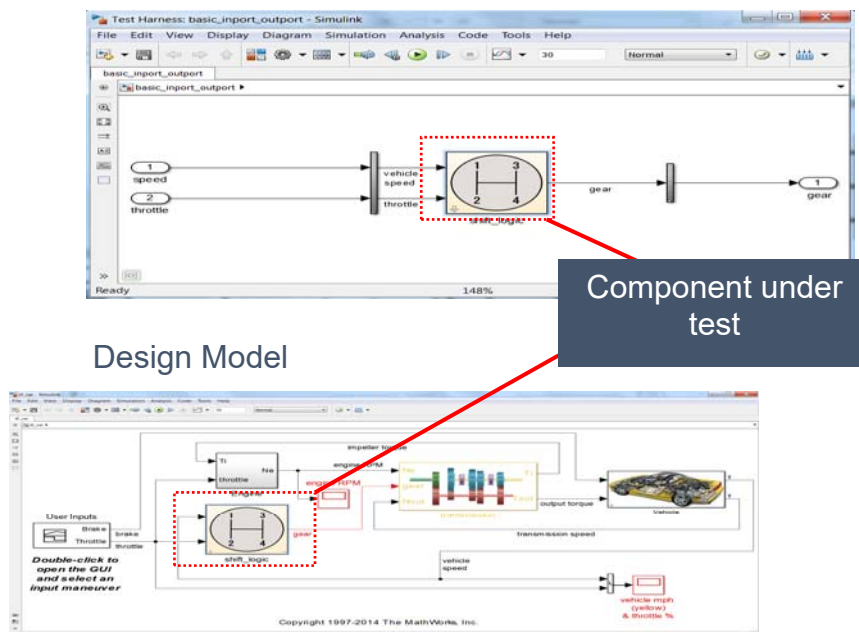


Figure 11: Test Harness and Design Model interaction [54]

Test Sequence/Assessment: Test Sequence/Assessment Blocks can be used for preparing reactive and/or time-based test cases. A test sequence consists of test steps arranged in a hierarchy. Test steps that are temporal or logic based provide value assignments to the inputs to the model under test. Additionally, each test step has the transition conditions, next step number and the description of the test step. ‘Transition’ specifies the condition on which the execution has to switch to the next test step. Timing relations between test steps are realized by using the ‘after(<n>,tick)’ function in the Transition conditions. The steps are sequentially executed in the order specified in the Test Sequence blocks.

Test Assessment block also has test steps, transition conditions, next steps and description fields. Test Assessment block has verify statements to verify the actual output values from model under test against the expected values. A test case fails if any of the verify statements in any of the assessment steps for that particular test case fails. 'Asserts' if used instead of 'verify' statements will stop the test execution whenever an assert expression falsifies.

5.6 Test Oracle Specification

Specifying the oracle information which is the expected outputs to assess the correctness of the model, can be achieved in several ways. Figure 12 shows, three different techniques to evaluate the correctness of the model behavior namely Simulation Based, Equivalence and Baseline testing.

Simulation based testing is the most common technique wherein we analyze the requirements and model to compute the expected output values when a given set of test input vectors stimulate the model. In Simulink Mathworks world, the assessment criteria is embedded using verify or assert statements within the model or in separate Test Assessment blocks.

Baseline Testing is the second technique wherein the actual outputs from the model are compared to a baseline dataset of expected outputs. In Simulink environment, the expected output values are in .mat or .xls file and are imported as the baseline criteria for testing.

Equivalence testing is the third oracle specification technique, where in the correctness of a model is decided by comparing with a reference model. Inputs are provided to stimulate both the models concurrently and the outputs from the models are compared over the entire time of test execution.

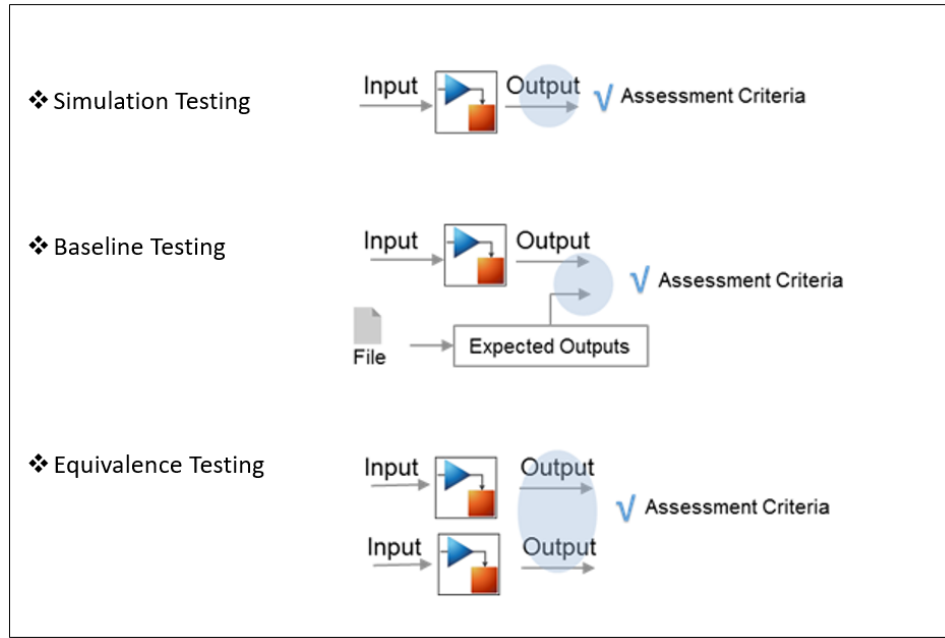


Figure 12: Simulation, Baseline and Equivalence Testing on Models [54]

5.7 Test Inputs Specification

The test case specification contains information on the purpose of testing a system, how to test a system, how to specify inputs, what all scenarios to test, and so on. The test procedure contains the details of sequence and values of test inputs and the timing of feeding them into the model component.

For testing real-time systems we need to target both the temporal domain as well as the value domain. Testing in the temporal domain has several implications. In some cases, the system design would be more computational and combinational logic, that it would be important to test the response of system to different combinations of input values. In other cases, the system design could be based on state machines and state transitions based on input events, and so it would be important to test the temporal response of system to different sequences of input events.

5.7.1 Sequence Based Testing

Sequence based testing can be defined as a time-based test-sequence driven testing process where testing concentrates on ensuring that with a stimulus input sequence, the right reactions (state change/output change) happen at the right time. In the target test system, SymPLe architecture, there were several components that were subjected to sequence tests. The Global sequencer, Local Sequencer and FB Controller are the three major components that together defines the internal state of the system. Global sequencer is the master component that schedules the multiple task lanes and handles IO data sharing between them, local sequencer handles the activation of the function blocks based on the sequence provided by the user, function block controller handles the state of execution individual function blocks. All these components have state machines implemented in them and the requirements specify the component behavior in each of its states and sub states. Hence testing these components mainly involves testing the state machines in these components. To ensure that components enter and exit parent or sub states correctly and at the right time, test cases needs to first put them in an initial state and then stimulate the transitions with input events. Thus the test cases shall involve a sequence of test steps to transition the component through the various states and shall evaluate the state behavior of the model with respect to time. In this way, sequence based tests help in verifying the temporal behavior of the model component. For example, in Local Sequencer, to test the behavior of the sub states in write state shown in Figure 13, we need to provide input values such that the execution first reaches FB_EXECUTE state. Further new_state variable becoming true shall result in Local Sequencer moving to the WRITE state. Thus our testcase involves multiple test steps, for traversing from INIT to WRITE state and also an assessment of the timing of state change and the output values once it enters the WRITE state.

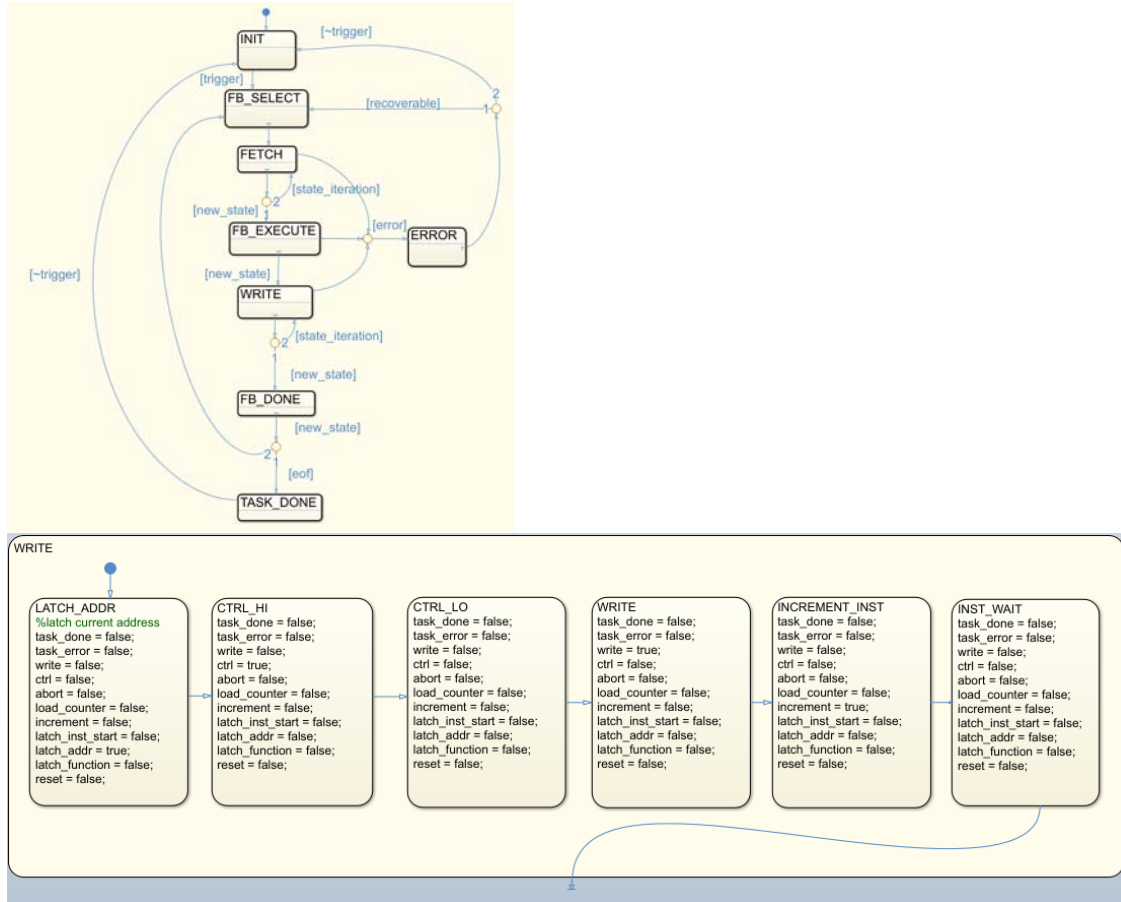


Figure 13: Local Sequencer Top level State Machine and sub-states in WRITE state

As we divide the testcases based on requirements that we are testing, we would also need to divide the test sequences and assessments based on what testcase is being executed. For example testcase that tests the FETCH state need to only verify the output value behavior until the component reaches FETCH state from INIT. Another testcase that tests WRITE state needs to verify the state behavior and output values in WRITE state as well. In this case, it is needed to continue the test steps and assessments after the component passes FETCH state. In order to assign the necessary test steps and assessments for the different testcases, TestCase parameter checks are added before verify statements and in the Next Step transition conditions as shown in Figure 17 and Figure 18. The TestCase parameter is set to values 1,2,3,4... based on the testcase, by specifying the command in the Pre-Load callbacks in Test Manager for each testcase, as shown in Figure 14. This Pre-Load callback commands are automatically run before the testcase starts executing. Based on the value set to TestCase parameter, only the required test steps for that testcase are run in the Test Sequence and only the necessary verify steps in the Test Assessment blocks are assessed. In this manner, by writing test input sequences in the Test Sequence block and verify

statements at the expected cycle time in the Test Assessment block, sequence based tests verify the timing behavior of outputs in the model

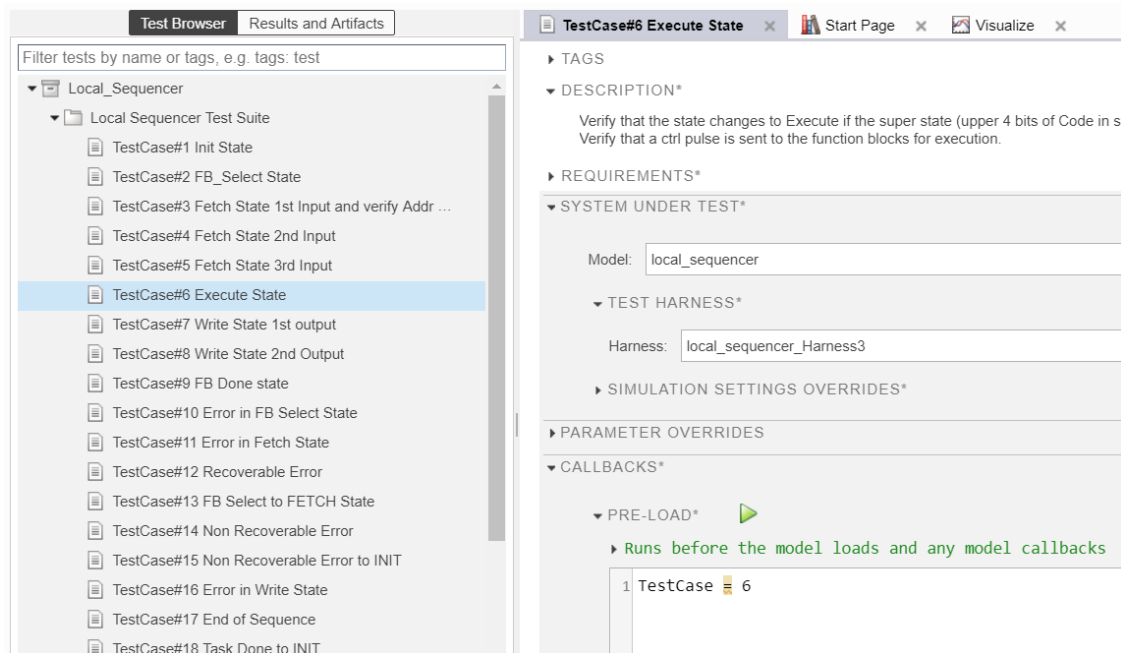


Figure 14: Assigning TestCase parameter value in each Testcase in Test Manager

Example: Local Sequencer Test Harness

The Local Sequencer Test harness is as given below in Figure 15. It shows the 3 inputs that needs to be fed into the Local Sequencer. In the architecture ‘trigger’ signal to the Local Sequencer actually comes from Global Sequencer, ‘inst’ from the sequence memory and ‘status’ from Function block Controller. As this harness is for unit testing of Local Sequencer these external inputs are now fed from the Test Sequence block. And all the outputs from Local Sequencer go into the Test Assessment block for their assessment and comparison to the expected output values.

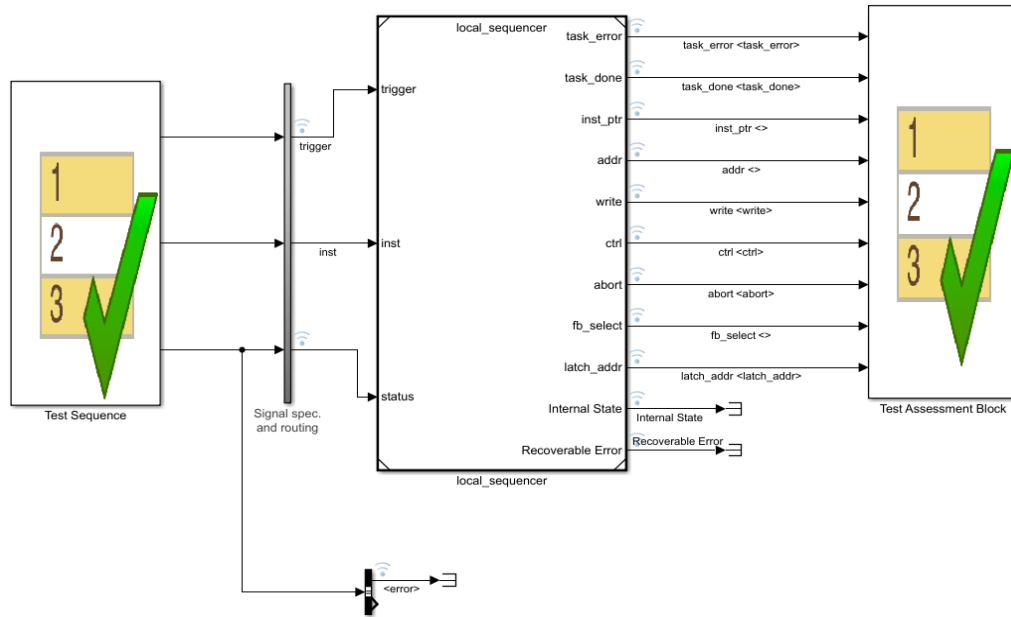


Figure 15: Local Sequencer Test Harness

There are 32 testcases in the Local Sequencer Test suite each having different purposes. The testcases are created and managed in Test Manager. The description of the testcases are given for each testcase as shown in below Figure 16.

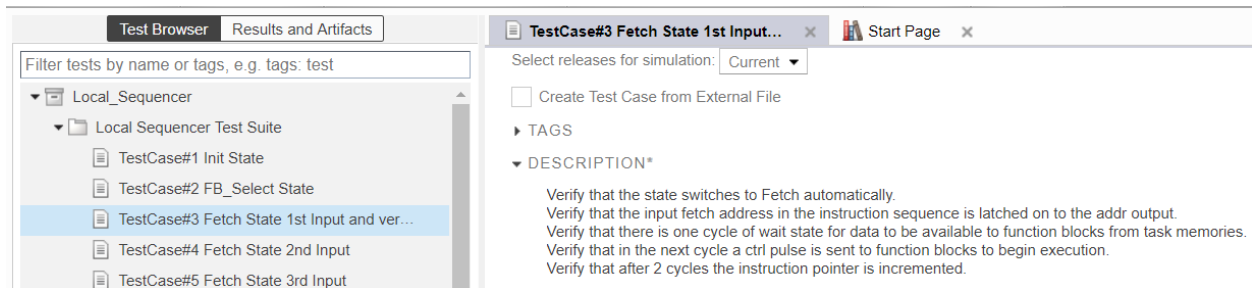


Figure 16: Test Case Description in Test Manager

Test steps provide value assignments to the three input signals 'trigger', 'inst' and 'status' to Local Sequencer. Below figures, Figure 17 and Figure 18 gives the Test Sequence block and Test Assessment in Local Sequencer respectively and it serves as a typical example of Sequence based testing. As can be seen in Figure 17, inputs are assigned with valid values in various steps and transition to the next step is based

on completion of cycle ticks and the TestCase parameter value. The description of each test step is given under the Description field. Similarly, in Figure 18, it can be seen that, verify statements are written in order to verify that, after the system has been stimulated with an input sequence, it has ended up in the correct state with correct output values within a defined number of cycle ticks.

| Step | Transition | Next Step | Description |
|--|---|-----------------------------|--|
| Init %% Initialize data outputs. trigger = false; inst = 26; status.done = false; status.busy = false; status.error = false; status.code = uint32(0); | 1. after(1,tick) && TestCase == 1 2. after(1,tick) && TestCase > 1 | EndTest step_1 | Init State input values. |
| step_1 trigger = true; | 1. after(1,tick) | step_2 | Trigger signal from Global Sequencer is made True |
| step_2 inst = 26; | 1. after(2,tick) && TestCase > 2 2. after(1,tick) && TestCase == 2 | step_3 EndTest | Set inst to 26 indicating the BitShift Left function block ID |
| step_3 inst = 32; status.code = uint32(0); | 1. after(6,tick) && TestCase > 3 2. after(6,tick) && TestCase == 3 | step_4 EndTest | Set the next instruction to 32. This becomes the 8bit task memory address from which the first input for the function block has to be fetched from. |
| step_4 status.code = uint32(4194304); inst = 33; | 1. after(6, tick) && TestCase == 4 2. after(6, tick) && TestCase == 5 3. after(6, tick) && TestCase > 5 | EndTest step_5 step_6 | Set the code bits of the code such that the 22-27 bits increment by 1 indicating a 2nd input to fetch. Set the task memory address to fetch the second input from as 33. |
| step_5 status.code = uint32(8386608); inst = 34; | 1. after(8, tick) && TestCase == 5 | EndTest | Set the code bits of the code such that the 22-27 bits again increments by 1 indicating a 3rd input to fetch. Set the task memory address to fetch the third input from as 34. |
| step_6 inst = 22; status.code = uint32(268435456) | 1. after(2, tick) && TestCase > 6 2. after(2, tick) && TestCase == 6 | step_7 EndTest | Set the code bits such that the upper 4 bits change from the previous value indicating Local Sequencer to transition to the Execute. Set the first output store address to 22. |
| step_7 status.code = uint32(536870912) | 1. after(6, tick) && TestCase > 7 2. after(7, tick) && TestCase == 7 | step_34 EndTest | Set the code bits such that the upper four bits change value from previous value indicating Local sequencer to transition to the Write output state. |
| step_34 inst = 22; | 1. after(1, tick) | step_8 | Set the second output store address to 22. |

Figure 17: Test Input Sequences in the Test Sequence Block

| Step | Transition | Next Step | Description |
|--|---|------------------|--|
| Run # TestCase == 1 verify(task_done == 0); verify(inst_ptr == 0); verify(task_error == 0); verify(latch_addr == 0); verify(write == 0); verify(ctrl == 0); verify(abort == 0); verify(fb_select == 0); end | 1. after(3,sec) && TestCase > 1 2. after(3,sec) && (TestCase == 1) | step2 EndTest | Verify that in Init state all the outputs are reset to 0. The instruction pointer should be reset to 0 to point to the first instruction in the sequence. |
| step2 # TestCase == 2 verify(task_done == 0); verify(inst_ptr == 0); verify(task_error == 0); verify(latch_addr == 0); verify(write == 0); verify(ctrl == 0); verify(abort == 0); verify(fb_select == 42); end | 1. after(1,sec) | step3 | Verify that with a trigger from global sequencer, the task execution is initiated. The state transitions from INIT to FB_SELECT. Verify that the function number register and start instruction pointer register are latched to the FB_SELECT. |
| step3 # TestCase == 2 verify(task_done == 0); verify(inst_ptr == 1); verify(task_error == 0); verify(latch_addr == 0); verify(write == 0); verify(ctrl == 0); verify(abort == 0); verify(fb_select == 42); end | 1. after(1,sec) && TestCase > 2 2. after(1,sec) && TestCase == 2 | step4 EndTest | Verify that the instruction pointer increments. |
| step4 # TestCase == 3 verify(task_done == 0); verify(inst_ptr == 1); verify(task_error == 0); | 1. after(1,sec) | step5 | Verify that the state switches to Fetch automatically. Verify that the fetch address in the instruction sequence is latched on the addr output. |

Figure 18: Verify statements in Test Assessment Block

5.7.2 Boundary Value Combinatorial testing:

This testing process ensures that the right output values are always calculated by the component algorithm with different combinations of sample input values chosen during equivalence partitioning and boundary value analysis. Boundary value analysis helps in reducing the input domain into a tractable set of input values. The combinatorial method, involves selecting test cases that cover the different t-tuple combinations of input parameters[58]. This combinatorial test method covering all combinations of the chosen sample input values, gives more priority to the verification of the value of outputs rather than the time of appearance of the output. So it can be considered as a logical behavior testing than temporal behavior testing. This method of testing is mainly applied to units or models, which has more of logical and mathematical operations than state machine logic. This testing strategy is for testing the computational logic within function blocks. There are several logical function blocks like NOR, AND, NOT, OR, XOR, NAND, several bitwise function blocks like BNOR, BAND, BOR, BXOR, BNAND, comparison function blocks like MAX, MIN, GT, GE, EQ, LT, LE, NE, arithmetic function blocks like ADD, SUB, MUL, DIV, bit shift function blocks BSL, BSR and other function blocks like MOV and MUX. These function blocks handles multiple datatypes Integer, Boolean, Safe Bool and Qmn fixed point datatypes. The function blocks also have multiple fault tolerance features implemented for detecting input or output errors like datatype errors, divide by zero, input overflow, output overflow and underflow errors. To verify the correct functionality of the function blocks, their multiple datatype support and invalid input and output detection capabilities, we need to provide different combinations of input datatypes and values to the function block modules under test.

Further to assess the computations and logic in function blocks, a M-script is used to feed in inputs and expected outputs as arrays and retrieve results in tabular format that enables easier comparison and analysis of results. Figure 19 shows an example of the test input and expected output array format prepared for 'Greater than' (GRT) function block. The four columns in the 'Inputs' array represents the Input1 datatype, Input1 value, Input2 datatype and Input2 value. Each row represents separate testcases. The comments gives description of the input values provided for each testcase.

```

Inputs = int32([67108864,674,67108864,673; % Testcase1 : Int type, greater int part, val1 and val2 +ve
67108864, 673, 67108864,674;% Testcase2 : Int type, lesser int part, val1 and val2 +ve
67108864,-2147483647, 67108864,-2147483648;% Testcase3: Int type, greater int part, val1 and val2 -ve
67108864,-2147483648, 67108864, 2147483647;% Testcase4: Int type, lesser int part, val1 -ve and val2 +ve
142606337, 2147483647, 142606336, 2147483647;% Testcase5: Qmn type, same +ve int part, but greater fractional part
142606337,-2147483647, 142606336,-2147483647;% Testcase6: Qmn type, same -ve int part, but greater fractional part
142606337,-2147483647, 142606337,-2147483648; % Testcase7: Qmn type, greater -ve int part, but same fractional part
142606337, 6000,134217728,7000; % Testcase8 : Qmn type, lesser int part, but greater fractional part
142606337, 0, 134217728,0; % Testcase9: Qmn type, same int part, but greater fractional part
142606337, 1, 142606337, 1;% Testcase10: Qmn type, same values
67108864,-2147483648, 67108864,-2147483648; % Testcase11: Int type, same values
67108864,-2147483647, 142606337,-2147483648; % Testcase12: Different datatypes for value1 and 2 Int type for val1
142606337,-2147483647, 67108864,-2147483648; % Testcase13: Different datatypes for value1 and 2, Qmn type for val1
268435456,-2147483647, 268435456,-2147483648; % Testcase14: Invalid datatypes for value1 and 2, Qmn type for val1
16777216,-2147483647, 67108864,-2147483648; % Testcase15: Different types for value1 and 2, Invalid type for val1
67108864,-2147483647, 33554432,-2147483648; % Testcase16: Different types for value1 and 2, Invalid type for val2

ExpectedOutputs = int32([16777216,1,0; %Testcase1
16777216,0,0; %Testcase2
16777216,1,0; %Testcase3
16777216,0,0; %Testcase4
16777216,1,0; %Testcase5
16777216,1,0; %Testcase6
16777216,1,0; %Testcase7
16777216,0,0; %Testcase8
16777216,1,0; %Testcase9
16777216,0,0; %Testcase10
16777216,0,0; %Testcase11
16777216,1,256; %Testcase12: Datatype Error
16777216,1,256; %Testcase13: Datatype Error
16777216,1,256; %Testcase14: Datatype Error
16777216,1,256; %Testcase15: Datatype Error
16777216,1,256; %Testcase16: Datatype Error
])

```

Figure 19: Feeding Test Inputs and Expected Outputs in an Array format within the M script

Similarly, another array is provided in the same m script with expected output values. The columns of the 'ExpectedOutputs' array represents the output datatype, output value and error code and the rows represents different test cases. The m script also runs the testcases after feeding in the input and expected output values to the Test Sequence and Test Assessment blocks and the results are stored into a table format as given below in Figure 20. This table format is easily readable and facilitates easier comparison of actual and expected results against the input values.

| Inputs | | | | ExpectedOutputs | | | ActualOutputs | | | Result Var1 |
|------------|-------------|------------|-------------|-----------------|-------|--------------|---------------|-------|--------------|----------------|
| Data Type1 | Value1 | Data Type2 | Value2 | Data Type | Value | FB ErrorCode | Data Type | Value | FB ErrorCode | |
| 67108864 | 674 | 67108864 | 673 | 16777216 | 1 | 0 | 16777216 | 1 | 0 | Passed |
| 67108864 | 673 | 67108864 | 674 | 16777216 | 0 | 0 | 16777216 | 0 | 0 | Passed |
| 67108864 | -2147483647 | 67108864 | -2147483648 | 16777216 | 1 | 0 | 16777216 | 1 | 0 | Passed |
| 67108864 | -2147483648 | 67108864 | 2147483647 | 16777216 | 0 | 0 | 16777216 | 0 | 0 | Passed |
| 142606337 | 2147483647 | 142606336 | 2147483647 | 16777216 | 1 | 0 | 16777216 | 1 | 0 | Passed |
| 142606337 | -2147483647 | 142606336 | -2147483647 | 16777216 | 1 | 0 | 16777216 | 1 | 0 | Passed |
| 142606337 | -2147483647 | 142606337 | -2147483648 | 16777216 | 1 | 0 | 16777216 | 1 | 0 | Passed |
| 142606337 | 6000 | 134217728 | 7000 | 16777216 | 0 | 0 | 16777216 | 0 | 0 | Passed |
| 142606337 | 0 | 134217728 | 0 | 16777216 | 1 | 0 | 16777216 | 1 | 0 | Passed |
| 142606337 | 1 | 142606337 | 1 | 16777216 | 0 | 0 | 16777216 | 0 | 0 | Passed |
| 67108864 | -2147483648 | 67108864 | -2147483648 | 16777216 | 0 | 0 | 16777216 | 0 | 0 | Passed |
| 67108864 | -2147483647 | 142606337 | -2147483648 | 16777216 | 1 | 256 | 16777216 | 1 | 256 | Passed |
| 142606337 | -2147483647 | 67108864 | -2147483648 | 16777216 | 1 | 256 | 16777216 | 1 | 256 | Passed |
| 268435456 | -2147483647 | 268435456 | -2147483648 | 16777216 | 1 | 256 | 16777216 | 1 | 256 | Passed |
| 16777216 | -2147483647 | 67108864 | -2147483648 | 16777216 | 1 | 256 | 16777216 | 1 | 256 | Passed |
| 67108864 | -2147483647 | 33554432 | -2147483648 | 16777216 | 1 | 256 | 16777216 | 1 | 256 | Passed |

Figure 20: Test Results generated by M Script - Inputs, Expected & Actual outputs for testcases in Tabular format

5.8 Test Execution

After the Test sequences and assessments are prepared, and the testcases are created in the Test manager, the entire test suite or individual testcases are executed in the Test Manager. As seen in Figure 21, the test execution results in a Pass or Failure. Figure 21 shows the test result summary of the Local Sequencer component. Two test cases failed out of the 32 testcases. TestCase Outcomes - Passed or Failed or Incomplete status of the testcases are shown in the Test Manager.

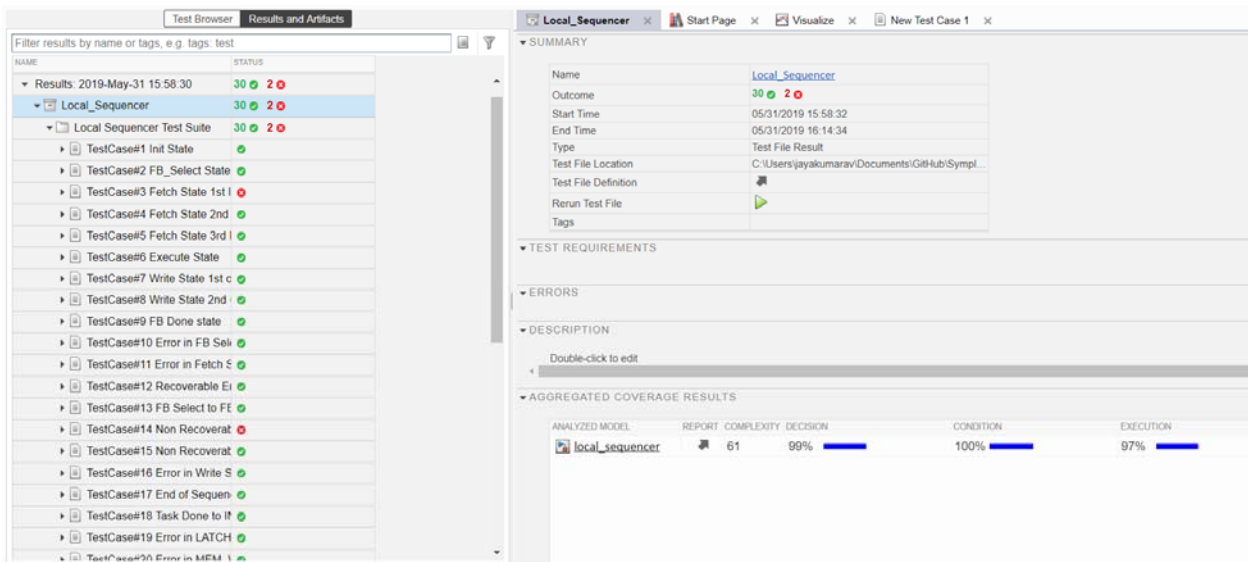


Figure 21: Test Results Summary in Test Manager

5.9 Test Results Analysis in MBD

After test executions, the failed test results are analyzed by graphically visualizing the failed test steps and signal behaviors. The analysis is intended to find the root cause for the test case failures. In situations, where the testcases are found to be incorrect or incomplete, they are updated and rerun. In other cases, the root cause is found to be a design issue in which case the design is reviewed along with the design team and the solution for the bug is figured out. In few cases, the root cause of the testcase failure is determined to be an incorrect requirement which would necessitate rewording of the requirements, which probably also leads to re-designing of the model and rewriting of the test cases. The testing process is thus an iterative process.

5.9.1 Analysis using Logic Analyzer

For analyzing the failed test cases, the verify statement is graphically visualized to analyze at what time instances the test steps are failing. The graph in Figure 22 shows that at time instances 54 to 58 ‘task_error == 1’ check is failing. Which means that ‘task_error’ signal was expected to be set high between time instances 54-58, but during execution of test cases, the ‘task_error’ signal is seen to be set low during clock cycles 54-58. As shown in Figure 23, Logic Analyzer is used to visualize the behavior of signals during test execution. FB Controller sets the ‘error’ signal in the status message high, if an error is detected in any function block execution. If the ‘error’ signal reported to the Local Sequencer is non-recoverable, it causes the Local sequencer to signal an error in the execution of the task lane by setting the ‘task_error’ signal high. The ‘task_error’ signal is expected to remain high until the global sequencer acknowledges receiving the ‘task error’ by pulling the ‘trigger’ signal low.

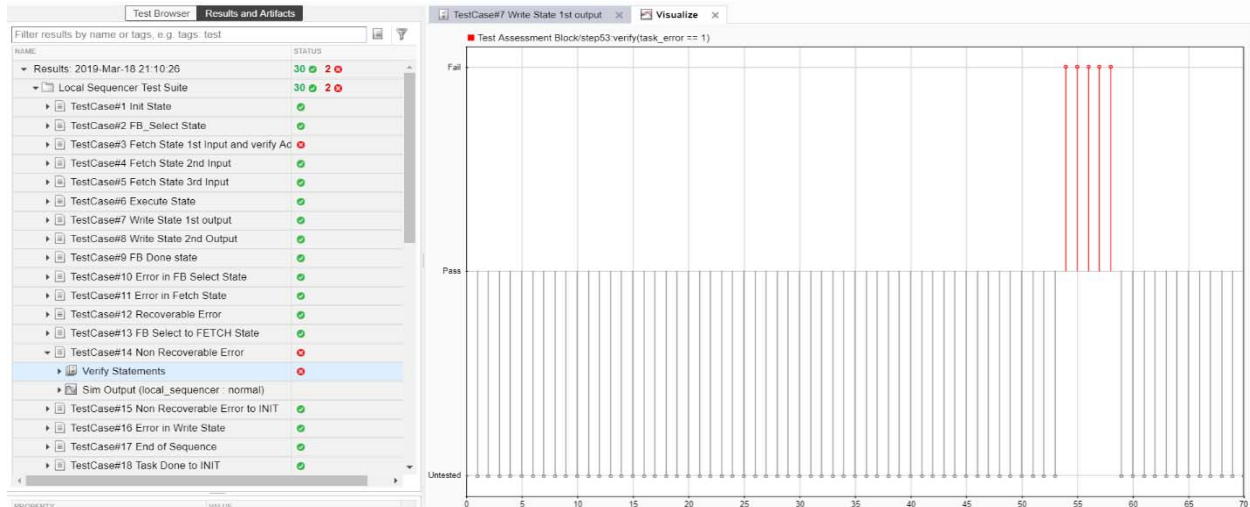


Figure 22: Graphical plot of Verify statement results with respect to test execution time in Test Manager

From the logic analyzer, it is seen that the ‘error’ input signal goes high while the Local Sequencer state is in EXECUTE state. The ‘Recoverable Error’ input is low during this time indicating that the error is non-recoverable. Hence Local Sequencer is expected to enter ERROR state and ‘task_error’ output signal is expected to be set high continuously while in ERROR state. But from the logic analyzer it is seen that during test execution, ‘task_error’ remains high only for one cycle and does not wait for the ‘trigger’ signal to be pulled low. This was noted as an incorrect behavior and indicates violation of requirements.

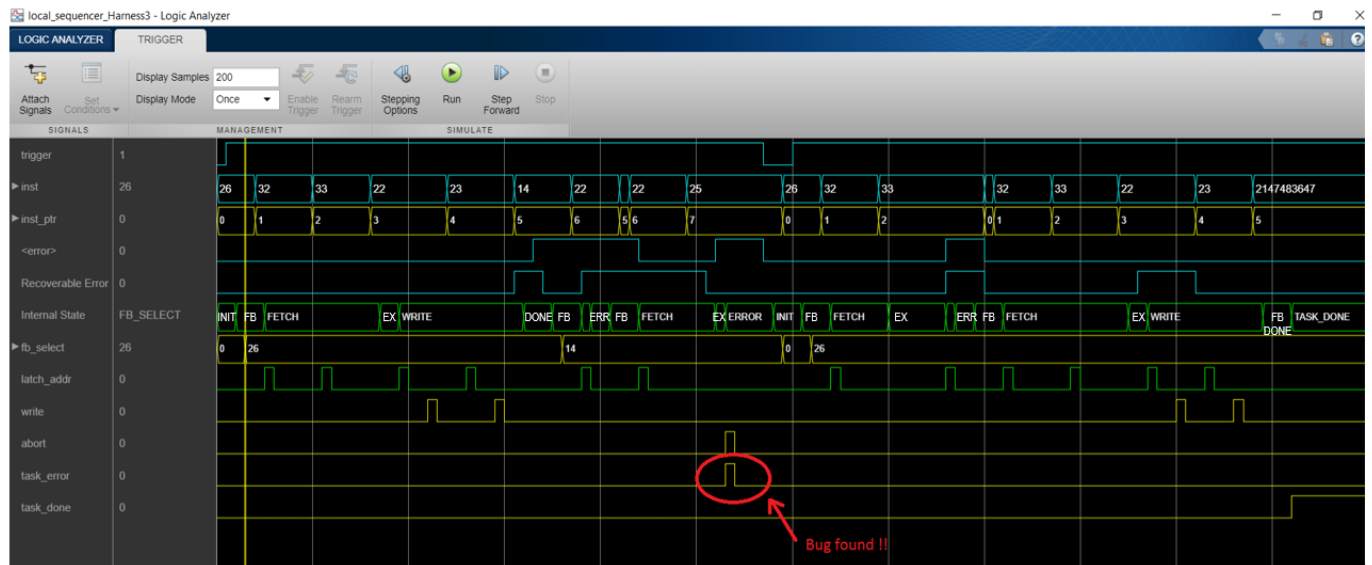


Figure 23: Analyzing Local Sequencer input and output signals in Logic Analyzer

On further analysis, it was found that the State Chart setting – “Initialize Outputs Every Time Chart Wakes Up” was checked for the state chart (Figure 25). Even though the statement ‘task_error = task_error’ in ABORT_Ctrl_LO state seems to be right, this assignment had no effect; ‘task_error’ signal was set to 0 in the ABORT_Ctrl_LO state due to the above mentioned property setting of the state chart. This state chart level property caused the chart to initialize all its outputs to the specified initial value whenever the chart wakes up which is basically in every cycle as this state chart is not under a triggered or enabled subsystem. Outputs are reset whenever a chart is triggered, whether by function call, edge trigger, or clock tick. Thus, though the ‘task_error’ signal was being set to ‘true’ in ABORT_CTRL_HI state, the signal value was not retained as ‘true’ in ABORT_CTRL_LO state, due to this state chart property.

Note: This indirect configuration-oriented bug, though challenging was caught by the model testing process well before HDL code generation.

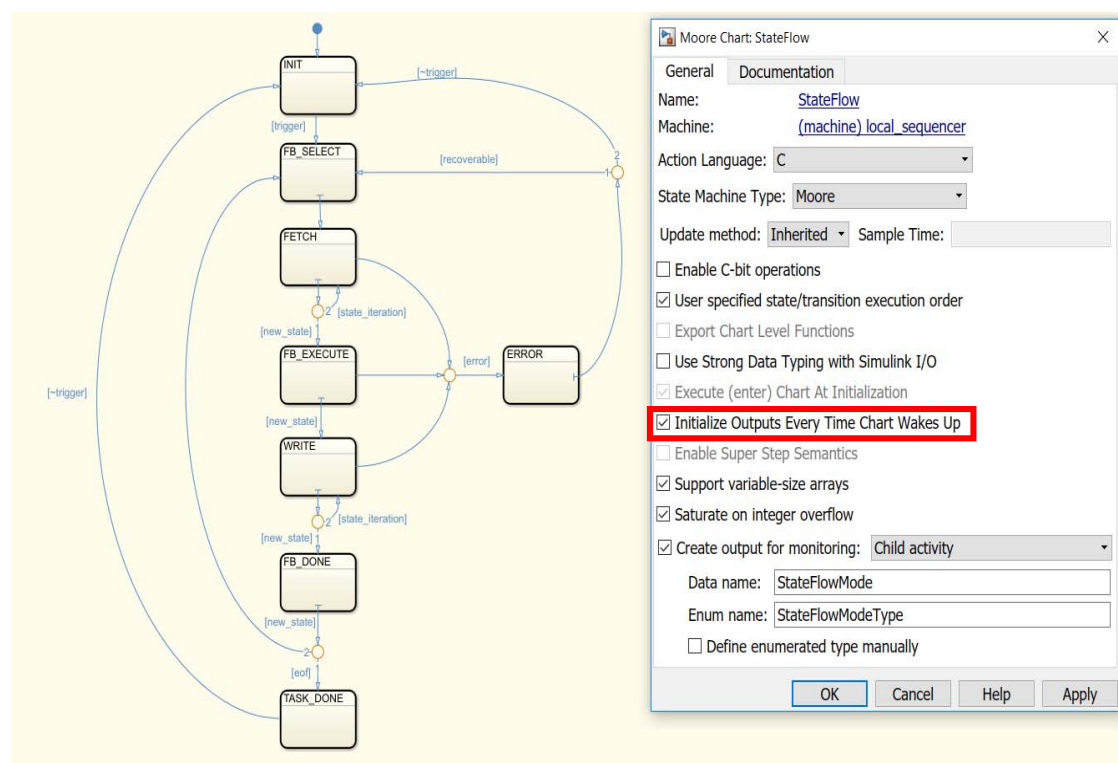


Figure 25: Local Sequencer State Chart Settings

5.9.2 Analysis using Animations

Animation of specification and design is a recommended technique for Software Verification in IEC-61508-3. The graphical animation feature in Simulink facilitates animating state charts to view the active and inactive states during debugging of a stateflow chart. Animations within Simulink have been found to be extremely helpful for the testers in analyzing some crucial test failures observed during Sequence based testing. One example is described here. Figure 26 shows the state machine in Function Block controller. The Function Block controller has various states INIT, IDLE, READ_INPUT, EXECUTE, WRITE_OUTPUT and DONE. WRITE_OUTPUT state will initiate writing output to an output address. Writing of output shall only happen after execution of the function block is completed. Completion of execution of function block is indicated by a signal 'executed'. Hence only when 'executed' signal goes high the function block controller shall transition from EXECUTE to WRITE_OUTPUT state.

From the logic analyzer view in Figure 26, it can be seen that the input 'executed' signal is always low during test execution and this shows that the execution of the function block is not completed during the entire test sequence. The Function block controller is expected to remain in the EXECUTE state until 'executed' signal goes high. But this expectation was not met and the related test case failed. Animation in Stateflow when turned ON, animates the state transitions when the testcase is executing. It can be seen that after few cycles of being in EXECUTE state, when 'ctrl' signal goes high, the state transitions to the WRITE_OUTPUT state, which is an incorrect behavior. This bug that was caught in the model testing phase is a critical bug, as the erroneous behavior seen could propagate to the system level, and it might end up in passing incomplete or incorrect results to the system output before the computations in the function blocks are actually completed.

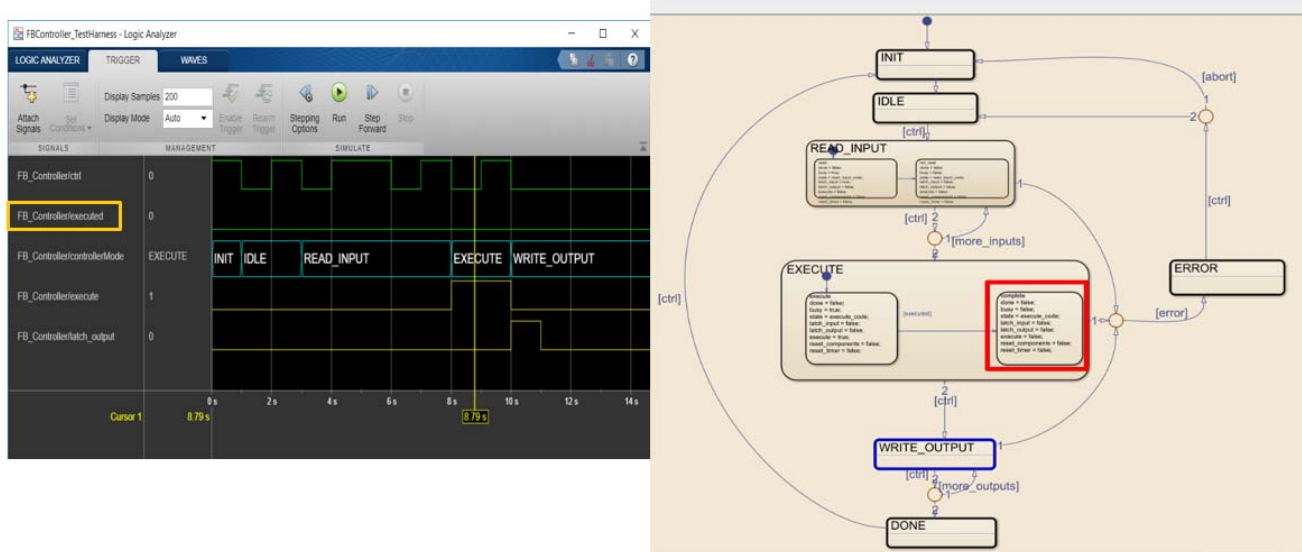
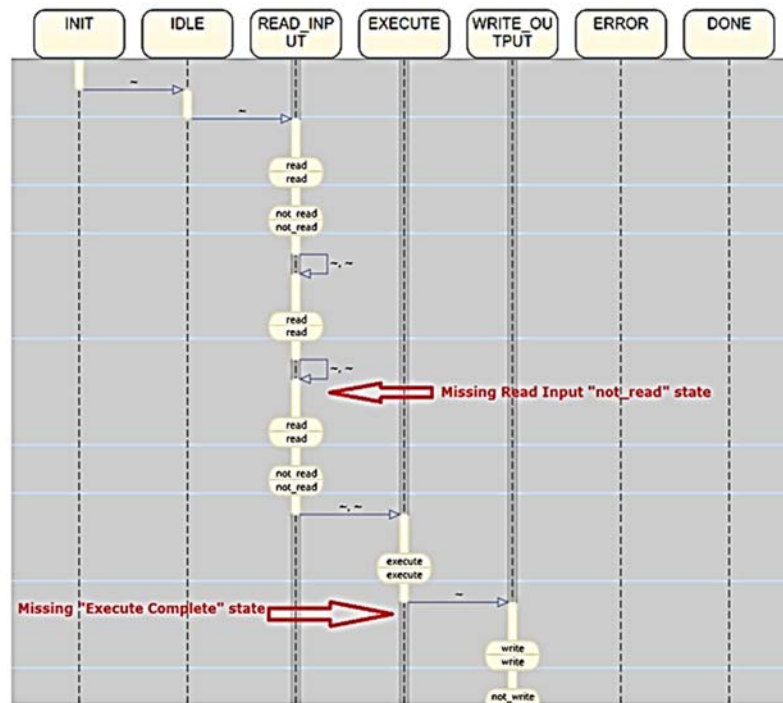


Figure 26: FB Controller Test Inputs in the Logic Analyzer and Animations in FB Controller State Machine

5.9.3 Analysis using State Sequences

The advantage of model based engineering is the variety of tools it offers for analyzing the dynamic behavior of the executable models. While animations offer a solution to dynamically view the state transitions during execution, State sequence analysis provides another option to statically view the sequences of state transitions during test execution. The sequences of state transitions and interchange of events and messages between stateflow charts can be captured in the ‘State sequence viewer’ in Simulink. State sequence analysis helps us to identify missing or wrong state transitions during test execution. Figure 27(i) shows the State sequence view of the Function Block Controller model with the above-mentioned bug. From State sequence view it can be seen that there is a missing state in between the ‘execute’ and ‘write’ sub states. This missing state is the ‘execute_complete’ sub state which is entered only when the ‘executed’ signal goes high. Ideally, the transition from the parent EXECUTE state to the WRITE_OUTPUT state shall only happen after it enters the ‘execute_complete’ sub state within the EXECUTE state and not while it is still in the ‘execute’ sub state (within the EXECUTE state). But here we see the transition to WRITE_OUTPUT state happening while the sub state is ‘execute’, i.e. Write output state is entered before function block execution is completed. The Figure 27(ii) shows the state sequence view after the bug in FB Controller component was fixed and the state sequence is now as expected. The FB Controller stays in the ‘execute’ substate waiting for ‘executed’ input signal to go high indicating the completion of function block execution.

State sequence with Bug



State sequence after Bug is fixed

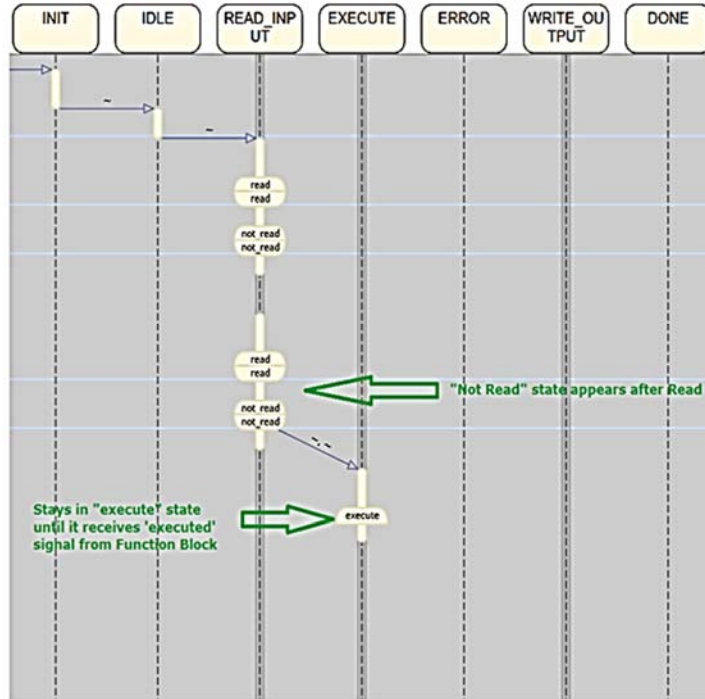


Figure 27: (i) State Sequence View with Bug in FB Controller, (ii) State Sequence View with Bug fixed in FB Controller

5.9.4 Analysis using Single step Execution

Interactive step by step simulations of the model played a major role in helping to understand the transfer and processing of data through the model. Single step execution is an execution mode supported in Simulink that allows tests to be executed stepwise which helps in visualizing the data getting propagated and computed within the model. Visualizing data values on the ports is enabled in Simulink by selecting the option “Show value label of selected port” for each signal line to be viewed. Figure 28 given below, is the screenshot of the Subtractor function block after several single step executions during test execution. The figure shows the data values on various points in the ‘Subtractor’ function block model after all the four inputs (2 input datatypes and 2 input values) have been read into the shift register in the function block. It can be seen that four inputs involving datatype and value of operand1 and datatype and value of operand2 are shifted into the Input registers. The input datatype 67108864 indicates Integer datatype. With input values being fed as 5000 and 4000, the output datatype is produced as Integer datatype as expected and the output value is also correctly calculated as 1000. The error code also indicates No Error (0) as anticipated.

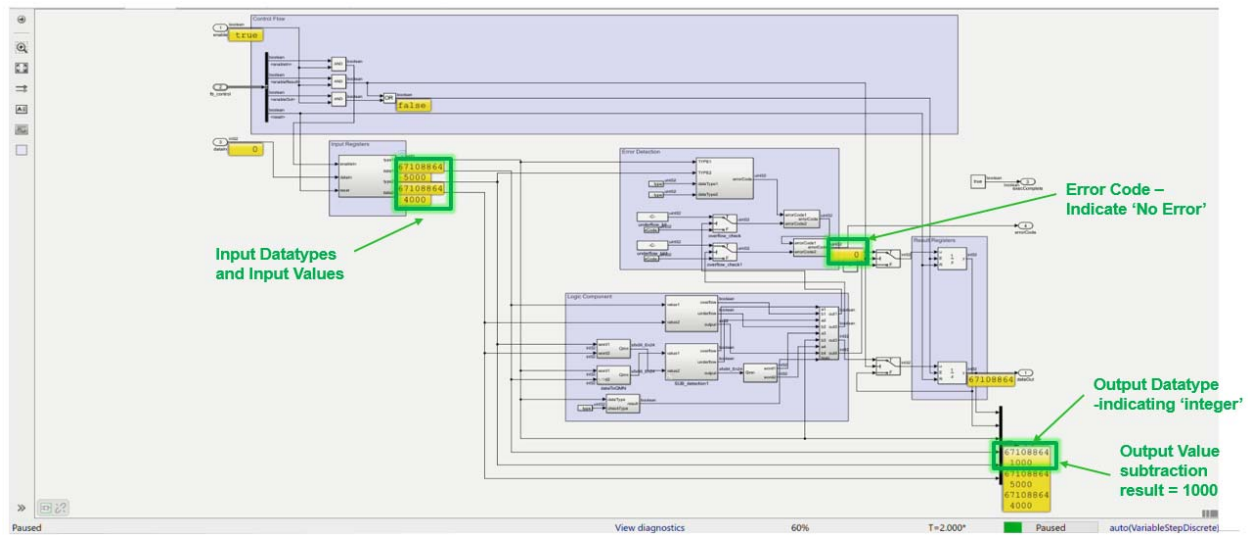


Figure 28: Single step execution in subtractor function block.

Datatype errors such as datatype mismatches between inputs and unsupported/invalid datatypes are detected by function blocks. The error code for datatype error is defined as 256 in the ‘setup.m’ configuration file (shown in Figure 29).

```

%FB ERROR Code Masks
mask.error.stateErrorBit = uint32(1);
mask.error.timeoutBit = uint32(2);
mask.error.overflowBit = uint32(4);
mask.error.underflowBit = uint32(8);
mask.error.divideByZeroBit = uint32(16);
mask.error.inputOverflowBit = uint32(32);
mask.error.outputOverflowBit = uint32(64);
mask.error.abortBit = uint32(128);
mask.error.typeBit = uint32(256);

```

Figure 29: Error Codes implemented in SymPLe.

The GRT(greater than) function block only accepts integer or Qmn type datatypes in both of its inputs. The Qmn datatype is a fixed-point datatype that consists of a 32-bit integer component and a 24-bit fractional component. The 24-bit fractional precision allows representation of data as small as 2^{-24} which is approximately 5.96046e-08. If the function block receives any other datatype other than integer or Q_{mn} as inputs, it is expected to indicate that with a datatype error of error code 256. Figure 30 shows the data value visualization in GRT function block during single step execution when a test with invalid input datatype is executed. It indicates that when the datatype of Input1 is the unsupported 'Boolean' type, it resulted in an error code 256 as expected.

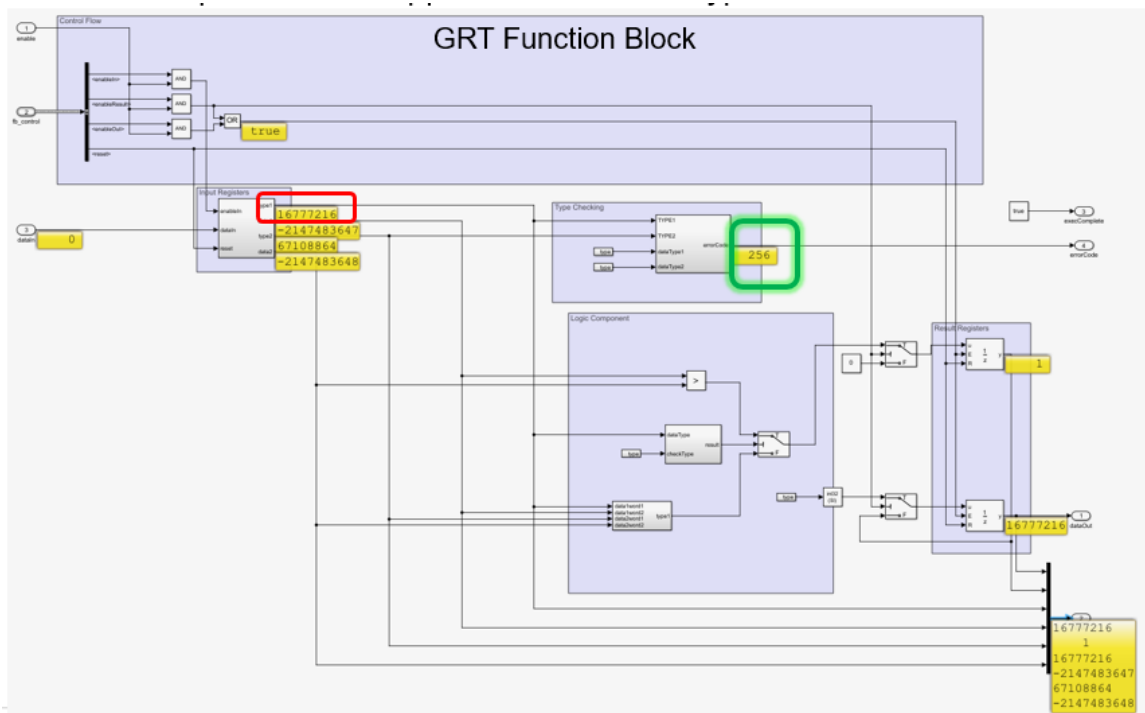


Figure 30: Datatype Error Detection

5.10 Integration Testing

After individually testing all the components in the SymPLe architecture, it is necessary to test the integration of all the components. In integration testing, we are interested in ensuring that the interfaces between components are fine and the signal flow and timing interactions between components work as expected. Complex interaction bugs are found in this phase. This level of testing prepares the base for the final System testing by confirming that each combination of modules within the system correctly work in unity. Integration testing can be considered to be a mixture of black box and white box testing.

To test the interaction of the local sequencer, function block controller and function blocks, tests were conducted at the task level. Test vectors were fed into a single task and the behavior of the components and data outputs were noted. The local sequencer and function block controller state machines had a tight connection such that the state change in one of the component drove the state change in other and vice versa. The 'ctrl' signal output from local sequencer triggers state progress in FB controller and the 'state' output signal from FBController drives state change in local sequencer. Hence the timing of state and output signal changes in both the components need to be consistent and perfectly designed so that the components can interact properly and progress within their state machines.

Tests at the task level, revealed that the local sequencer and FB Controller state machines enter into a deadlock situation as each of them are waiting for inputs from the other, as shown in Figure 31. Local sequencer is stuck in the FETCH state (Figure 32) and FB Controller is stuck in EXECUTE state and does not proceed any further from that. With few modifications in the stateflow design it was able to bring timing consistency between state changes in the two components. Thus, with the fixed design, the local sequencer state machine is seen to traverse through all states until it reaches the FB_DONE state (Figure 33).

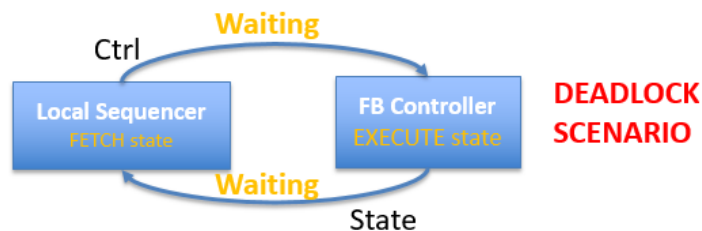


Figure 31: Deadlock Scenario detected in Integration testing

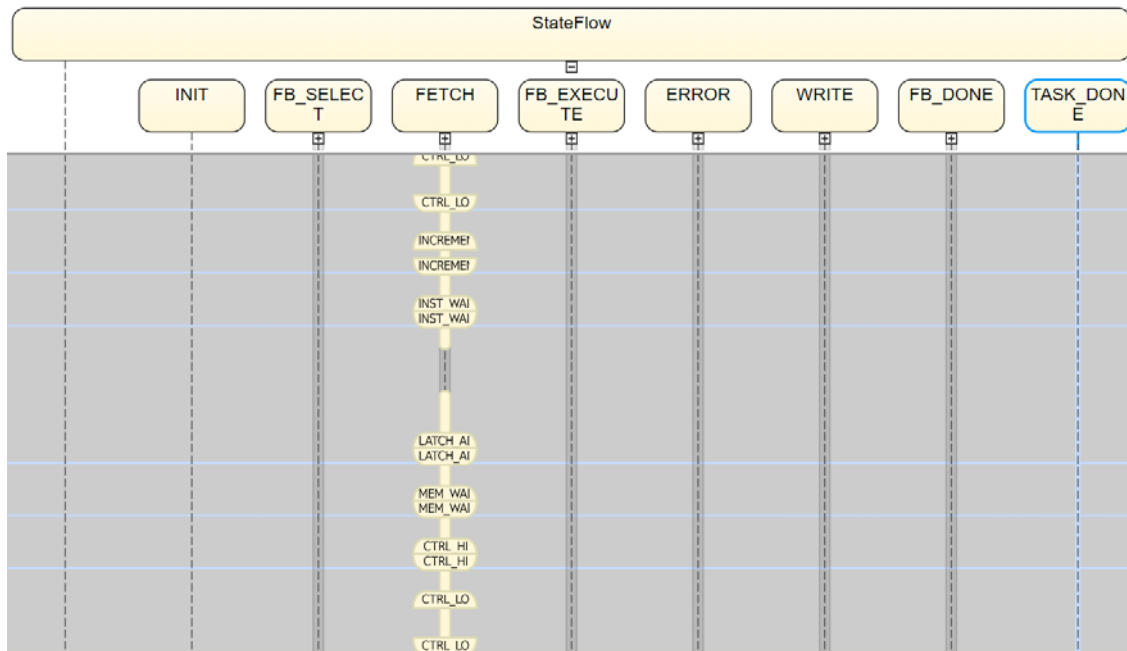


Figure 32: Integration Test Failure: Local Sequencer state sequence view

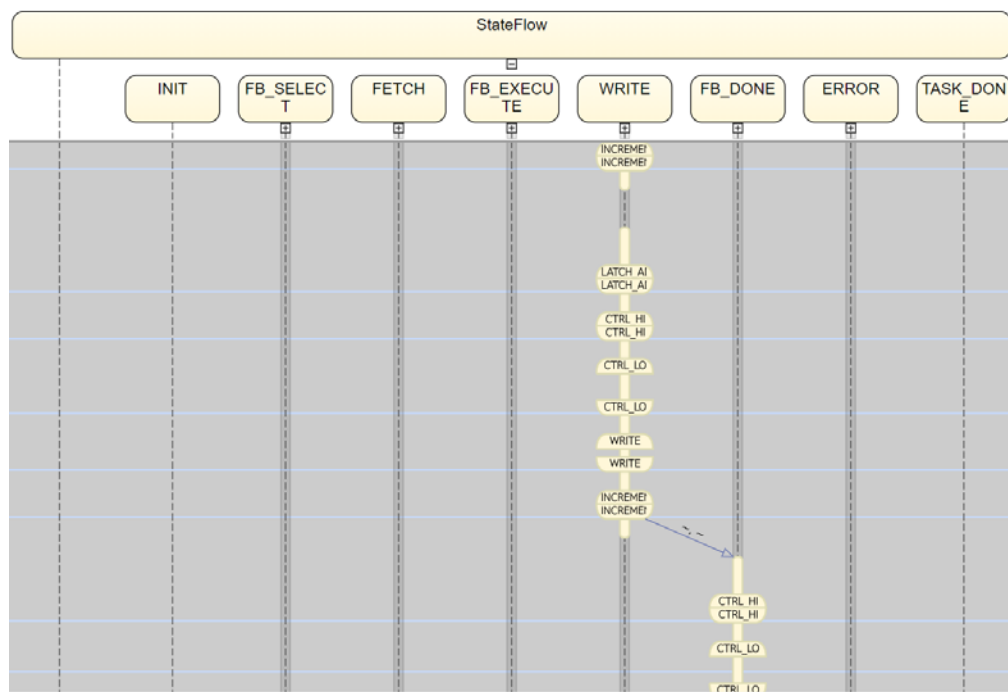


Figure 33: Local Sequencer State sequence view after Design fix

5.11 System Testing

System testing is the next level of testing in which the functionality of a completely integrated system is verified. All components are expected to have passed the unit and integration testing phases before proceeding to System testing phase. While integration testing at the model level tests the combinations of 2 or 3 components within the system model, system testing tests the top level model that contains all the model components/subsystems integrated together. System testing is black-box testing in which system inputs are fed and system outputs are verified for compliance to the system requirements.

Model-based system testing of this digital I&C architecture, SymPLe, is carried out by building a use case application using this architecture and then subjecting this application for black box testing. For this a Proportional Integral Derivative Controller (PID) application was implemented on the SymPLe architecture. The PID can be considered to be a composite function block made with elementary function blocks in SymPLe. The PID runs on one task lane in the SymPLe architecture. As shown in Figure 34, the Application testing is carried out in a complete Model-In-the-loop environment where the PID controller is analyzed along with a simulated plant environment. The controller output goes into the plant that is simulated using a transfer function and the plant output is further fed back into the PID controller. The Model-in-loop (MIL) test results are further considered as test references for comparison with subsequent non MIL tests, i.e. Software-in-the-loop and Hardware-in-the-loop tests.

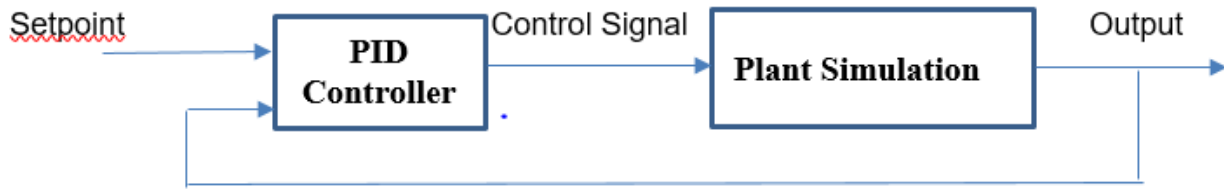


Figure 34: PID Controller and Plant Closed Loop Test Environment.

For testing the implementation of PID on SymPLe an oracle that is based on diverse implementation equivalence is used. As seen in Figure 35, two reference PID implementations are used for comparing with the PID implemented on SymPLe. A PID implemented with basic Simulink library blocks is one reference and the Discrete PID Controller Library block in Simulink library is the second reference. Both the reference PID controllers and SymPLe PID are arranged in closed loop with the exactly similar plant models. For the purpose of reference comparisons, an additional output signal 'PID_Trigger' is pulled out from SymPLe PID. The PID trigger is a Boolean output from SymPLe PID that goes high when the PID

output is written to the output address. This trigger signal is used to trigger the Reference PIDs so that all the PIDs update their outputs at the same time and thus enable simple output comparison. The PID implemented on SymPLe architecture updates the output after several ticks of simulation due to the multiple parent and sub states involved in GS, LS and FB Controller components, the combinational logic involved in the function blocks and their sequential activations based on the user-provided PID sequence in the sequence memory. Whereas, the reference PIDs are simple PID implementations that update new output value every simulation cycle. Hence, to synchronize the outputs from Reference PIDs and the output from PID on SymPLe we need to have this additional trigger signal.

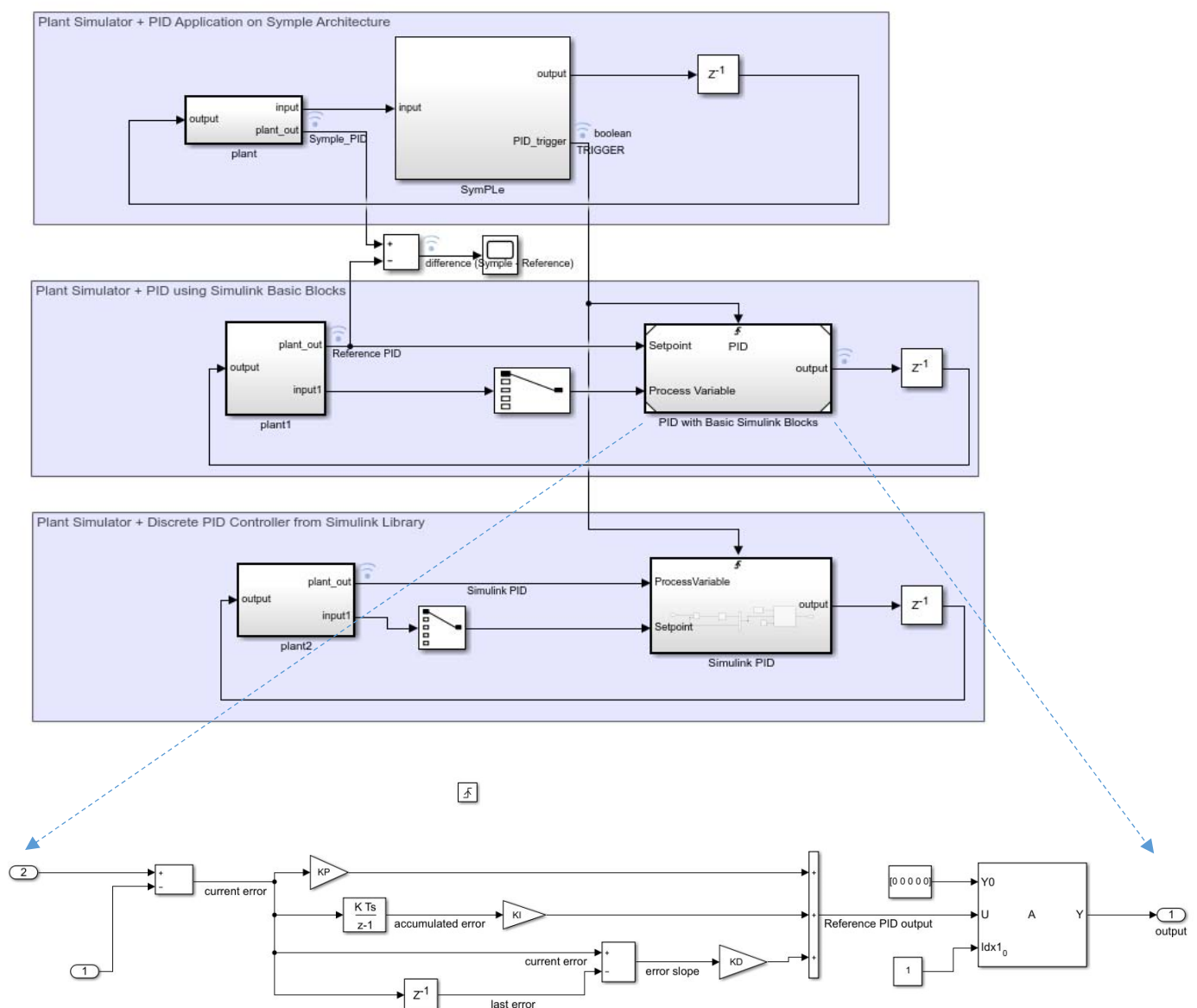


Figure 35: Equivalence Test Environment for PID Application on SymPLe architecture

Figure 36 shows the plant outputs obtained from the three closed-loop ‘PID controller-plant’ test setups. The purple line represents plant output with SymPLe PID controller, yellow line indicates plant output with the ready-to-use discrete time PID controller in Simulink library and green line indicates plant output with reference PID controller designed with Simulink basic blocks. The plant output with SymPLe PID controller closely matches the plant outputs with the Reference PID controllers. With transfer function of plant being set to ‘ $1/z$ ’, and the parameters setpoint = 50, k_D (Differential gain) = 0.2, k_P (Proportional gain) = 0.2, k_I (Integral gain) = 0.1, a critically damped plant output is obtained. The plant output slowly increases until it stabilizes at the specified setpoint 50.

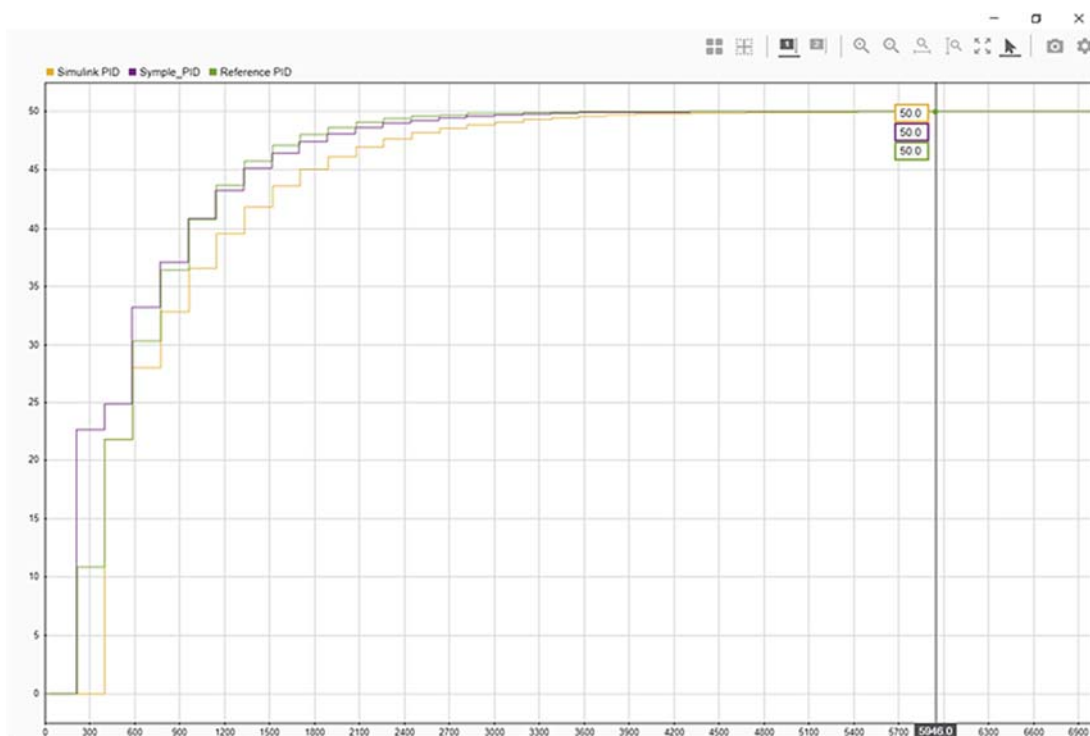


Figure 36: Graphical plots of PID outputs from SymPLe PID and the two reference PIDs in Simulink Data Inspector

Chapter 6 Model Coverage Analysis Workflow

This chapter describes the second process in the SymPLe verification workflow immediately following the Model testing phase, which is the Model Coverage Analysis. This chapter introduces the two classes of model coverage, which are Functional and Structural coverage and discusses the potential reasons for a low structural coverage on Models.

Coverage analysis is an integral part of testing and an inevitable process in the verification workflow. Model coverage is an important indicator to determine the completeness and quality of the Model testing phase. As shown in Figure 37, test coverage has two aspects Functional and Structural coverage. Model coverage is useful to find out gaps in requirements, design and testing and thereby prevent or reduce the leakage of defects that can happen during model testing phase. Aldrich in [59], introduced the usefulness of model coverage analysis for measuring Requirement completeness, Test generation and for demonstrating model and target equivalence through testcases that achieve the mandated coverage.

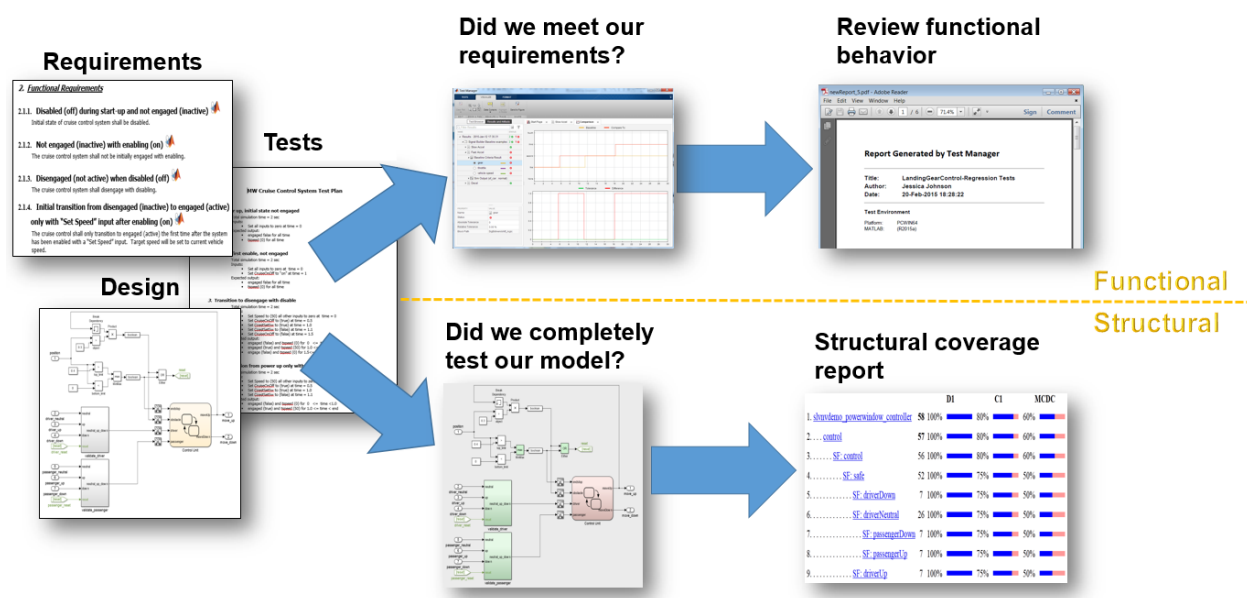






Figure 37: Functional and Structural Coverage [54]

6.1 Functional Coverage

Functional Coverage measures the completeness of covering the requirements with the test vectors. Exhaustive testing is expected to cover all the functional requirements for the model under test. Simulink provides an integrated Requirements Editor and possibility to link the testcases to the requirements by creating bi-directional links. Thus, we can trace from requirements to testcases and back from testcases to requirements through the forward and backward traceability links. There could be multiple requirements linked to a particular testcase when the test case verifies the model's compliance to multiple requirements. Also, multiple testcases can be linked to a requirement, when we use more than one testcase (sometimes in order to meet 100% MC/DC structural coverage) to verify a particular requirement. Figure 38 given below, shows the view of Requirements Editor in Simulink with verification status highlighted. Red verification status indicates 'Failed' Testcases. Green verification status indicates 'Passed' testcases and Blue verification status indicates 'not verified but justified' testcases. It shows 'Verified by' Forward Traceability Links that links the requirement to one or more testcases or property proofs. The implementation status indicates the implemented and not implemented requirements. The 'Implemented by' Links traces the requirement to the model object/block which implements that.

-  - Indicates Requirements that are tested and Passed
-  - Indicates Requirements that are tested but Failed
-  - Indicates Implemented Requirements.
-  - Indicates Requirements that are justified for verification and implementation.

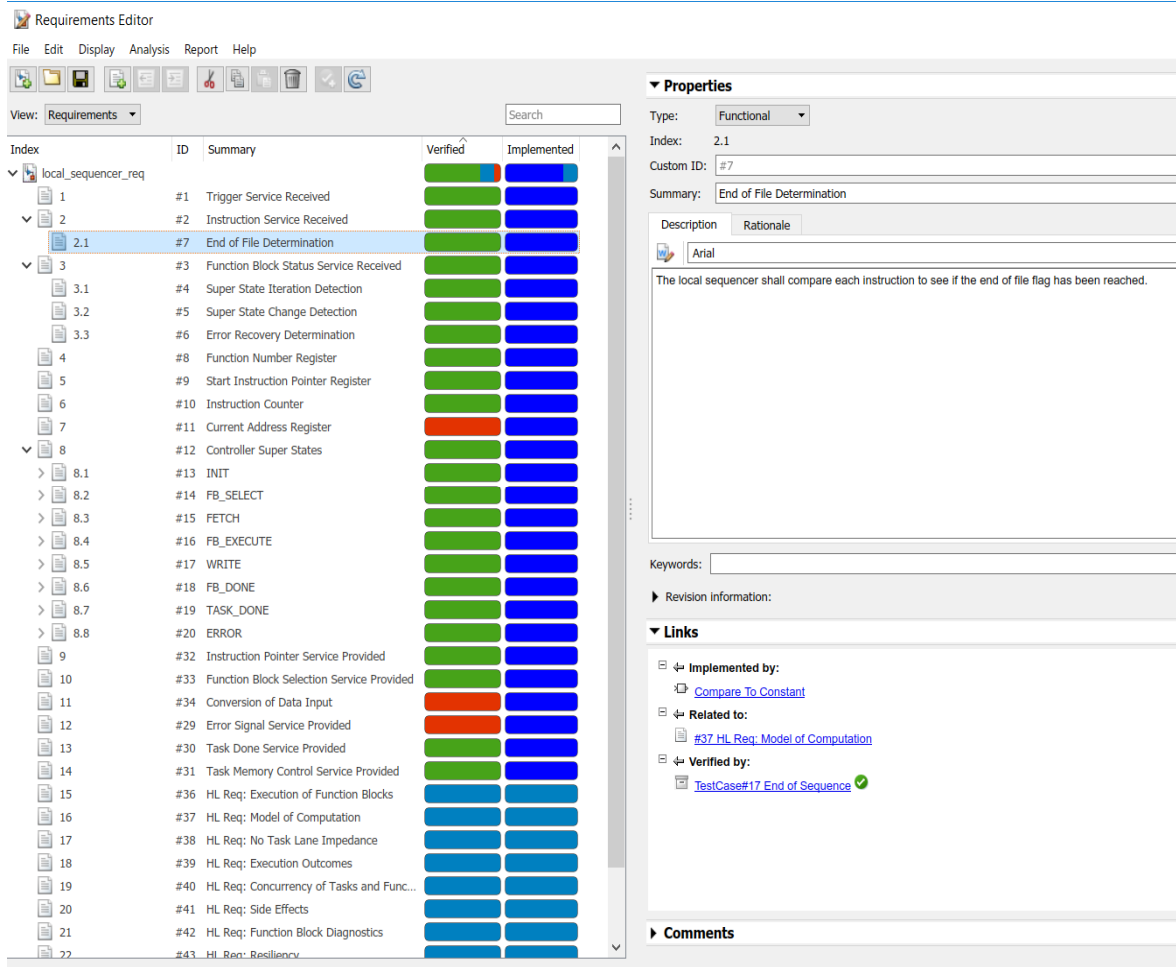


Figure 38: Requirements Editor view showing Implementation and Verification status and Forward traceability links from Requirements to testcases.

Similarly, Figure 39 shows the Test Manager with backward traceability links linking Testcases to the Requirements. It can be seen that 'TestCase 17 End of Sequence' is linked to four separate requirements.

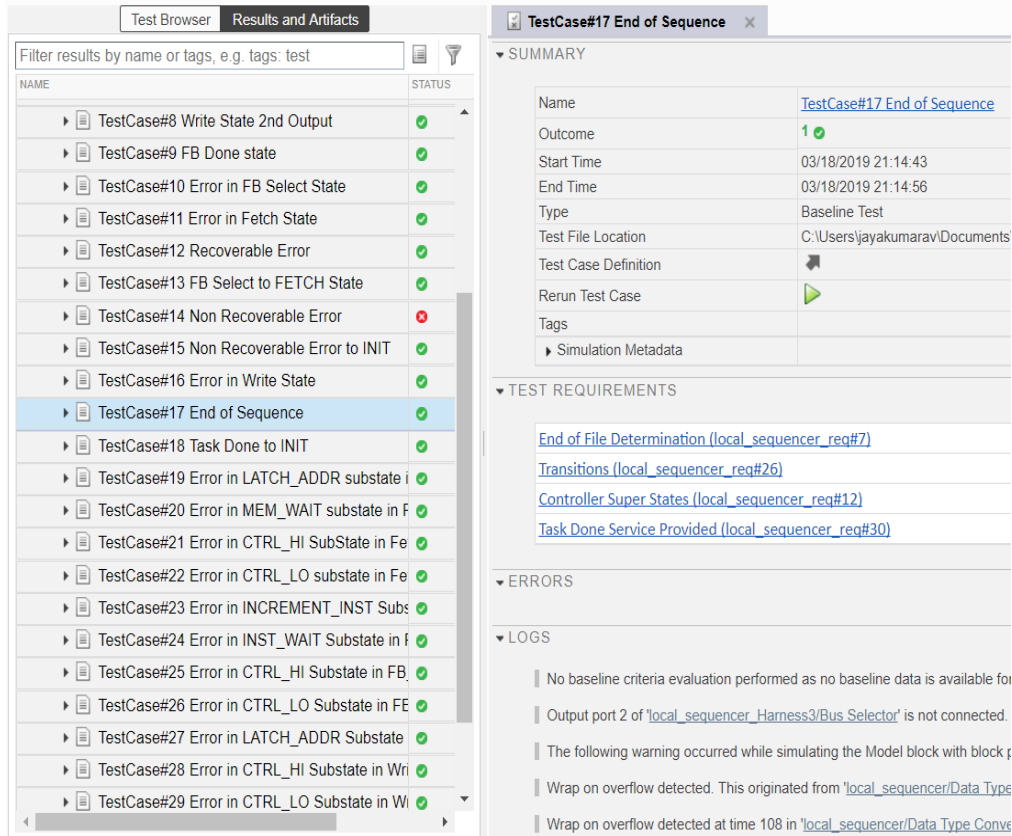


Figure 39: Test Manager view showing Backward Traceability links from Testcases to Requirements

6.2 Structural Coverage

Structural coverage measures the completeness of covering the model under test with the test vectors. Exhaustive testing is expected to traverse all the execution paths in the model. Simulink has inbuilt coverage analysis tool with advanced features.

The main motives behind measuring the test coverage are [54]:

- to determine dead logic branches
- to determine if sufficient test vectors have been created
- to determine if existing requirements are sufficient.

There are several types of coverage that can be measured on the models:

Execution Coverage - Execution coverage measures if an object/block in the model has been covered during the test executions. It determines whether a Simulink item, such as Simulink blocks, subsystems, Matlab functions, are executed atleast once during simulation. [60]

Decision Coverage - Decision coverage measures the percentage of all possible outcomes that a decision point takes during test executions. Decision elements in the Simulink Modeling environment include switch or Mux or If-Else blocks, or states.

Condition Coverage - Condition coverage measures if any condition/clause that is part of a decision/predicate and results in Boolean output, takes both the true and false path. Conditions usually involve logical operator blocks like Boolean AND, OR, NOT or relational operator blocks like Greater than, less than and other comparison operators or State transitions.

Modified Condition/Decision Coverage (MC/DC) – MC/DC (shown in Figure 40) is the most stringent type of coverage. MC/DC coverage demands that each condition in a decision shall independently change the outcome of the decision. This means that the each clause in a predicate shall be the only active clause causing false and true decisions during the test. MC/DC only applies to decisions with more than one condition. If there is only one condition/clause, MC/DC is equivalent to Decision coverage. For example, MC/DC coverage does not apply to Local Sequencer module as it does not have even a single multi condition decision point. All state transition conditions in Local Sequencer are simple single boolean conditions. Hence, MC/DC coverage is not applicable for the Local sequencer component and appears as Decision coverage in the Coverage report, as shown in Figure 42.

For a multi input block, full MC/DC coverage for a block is achieved when a change in one input, independent of any other inputs, causes a change in the block output.[60]

For multi condition transitions, full MC/DC coverage for a Stateflow transition is achieved when a change in each condition triggers the transition independent of the other condition.[60]

Condition
Decision
if (X && Y)
 Z = 1;
else
 Z = -1;
end

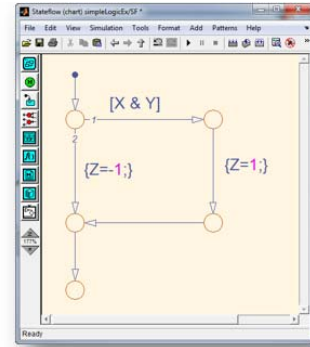
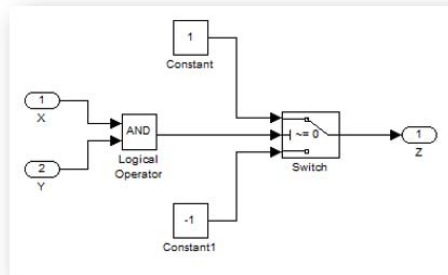


Figure 40: MC/DC Coverage [54]

Additionally supported coverage metrics in Simulink are Signal Range, Signal Size, Lookup Table and Relational boundary coverage. Signal Range coverage measures if signals in the model are covered for their entire valid range of values they can take. The signal range analysis gives the minimum and maximum values taken by the signals during the test execution. Signal Size coverage checks for size coverage for variable size signals. Lookup Table coverage measures how many lookup table values have been covered by the tests. Relational Boundary coverage measures the coverage of relational operators to see if they operate on equal operands and unequal operands that differ by a specific tolerance.

The MC/DC coverage was selected as the test coverage metric and it gets highlighted in the model which makes it easier to analyze the missing coverage paths and blocks. Figure 41 given below shows blocks and state machine with less than 100% test coverage highlighted in red. The state machine has 100% decision coverage which means all states in it are covered but has only 75% condition coverage and 50% MC/DC coverage indicating that some state transition conditions are not fully covered for both True and False paths.

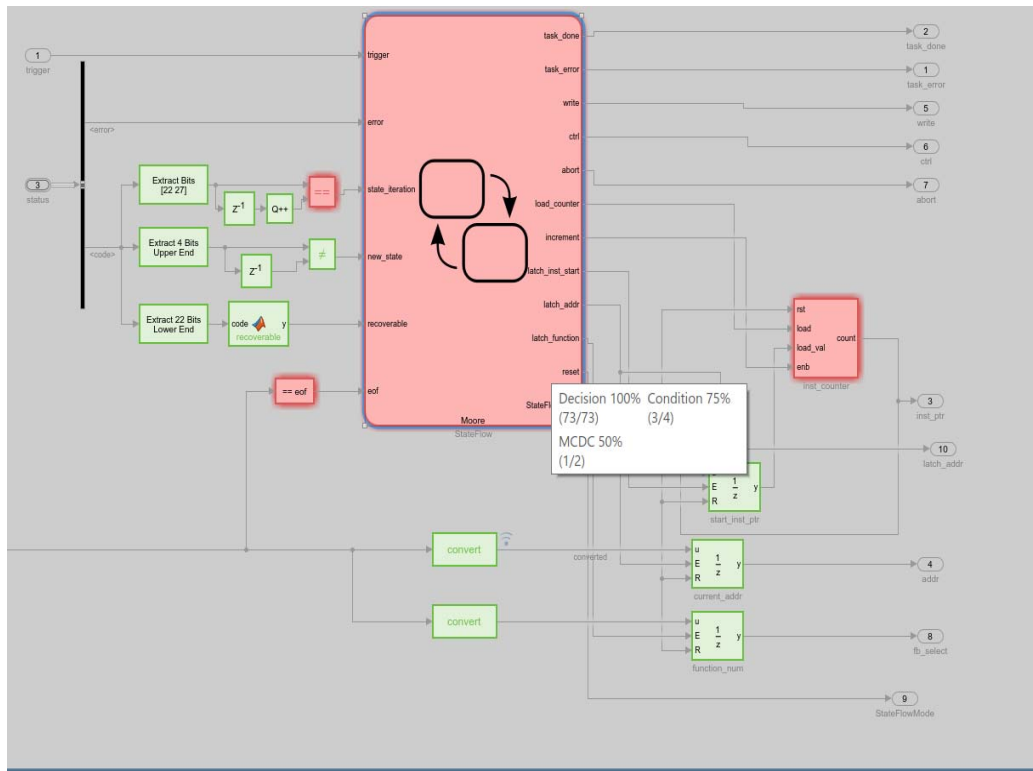


Figure 41: MC/DC Coverage highlighted in the model

A detailed model coverage report can be exported from the Test Manager that gives a good summary and details of the MC/DC, Execution, Condition and Decision coverage of test cases for each and every subsystem, block and state in the model under test. Figure 42 shows the Coverage summary in Local Sequencer model coverage report. An overall Execution coverage of 97%, Condition Coverage of 100% and Decision coverage of 99% is obtained. MC/DC coverage is not shown as it is not applicable for the Local Sequencer component indicating that there were no multiple condition decision statements or state transitions within the model. This depicts that the component design indeed satisfies the ‘constrained complexity’ design principle of SymPLe architecture.

Summary

| Model Hierarchy/Complexity | Test 1 | | |
|---|----------|-----------|-----------|
| | Decision | Condition | Execution |
| 1. local_sequencer | 61 99% | 100% | 97% |
| 2. . . . Compare To Constant | NA | 100% | 100% |
| 3. . . . Extract Bits | NA | NA | 100% |
| 4. . . . Extract Bits1 | NA | NA | 100% |
| 5. . . . Extract Bits2 | NA | NA | 100% |
| 6. . . . Increment Stored Integer | NA | NA | 100% |
| 7. . . . MATLAB Function | 3 100% | NA | NA |
| 8. . . . StateFlow | 50 100% | NA | NA |
| 9. SF: StateFlow | 49 100% | NA | NA |
| 10. SF: ERROR | 3 100% | NA | NA |
| 11. SF: FB_DONE | 1 100% | NA | NA |
| 12. SF: FB_EXECUTE | 2 100% | NA | NA |
| 13. SF: FB_SELECT | 4 100% | NA | NA |
| 14. SF: FETCH | 10 100% | NA | NA |
| 15. SF: WRITE | 10 100% | NA | NA |
| 16. . . . current_addr | 1 NA | NA | 100% |
| 17. Unit Delay Enabled Resettable | 1 NA | NA | 100% |
| 18. . . . function_num | 1 NA | NA | 100% |
| 19. Unit Delay Enabled Resettable | 1 NA | NA | 100% |
| 20. . . . inst_counter | 4 88% | NA | 93% |
| 21. Add_wrap | 1 50% | NA | 75% |
| 22. . . . start_inst_ptr | 1 NA | NA | 100% |
| 23. Unit Delay Enabled Resettable | 1 NA | NA | 100% |

Figure 42: Model Coverage Analysis summary

6.3 Reasons for Low Model Coverage

The potential causes of less than 100% structural coverage are missing requirements, over-specified design, design errors and missing tests. Tests may not fully cover all execution paths in the model or all values of conditions. Thus, less coverage could indicate missing tests. In other cases, some parts of the model are not covered by the tests as there are no requirements referring to them. Thus, missing requirements could be another cause for missing tests and therefore lower coverage. Alternatively, less coverage could indicate design errors like an unintentional dead logic in the model. Dead logic is a logic or part of the model that is never executed in any input conditions when the model executes. The other design related reason for less coverage is over-specified design. Over-specified design is a design that is specified with more conditions/clauses than what is necessary for intended behavior. With redundant conditions present in decision statements, it will not be possible to cover all combinations of conditions

Chapter 7 Static Verification

This chapter describes the Static Verification process which is the third verification technique applied after the Model Testing and Coverage Analysis processes. This chapter describes the details of the two separate static analysis checks performed on the SymPLe architecture (1) Conformance to IEC 61508 standard (2) Design Error Detection.

Static Verification is the process of statically analyzing the model and verifying if it adheres to modelling guidelines and various industry safety standards, without executing the model. Model Advisor in Simulink is used to statically check for design errors and model settings that cause generation of inefficient code or code unsuitable for safety-critical applications.

7.1 Static Conformance to IEC 61508

Using model advisor we check if the models conform to IEC 61508 standard. There are several safety relevant checks related to Simulink blocks, state machines and MATLAB code in the model implementation. Additionally, the IEC 61508 static verification also checks if the configuration settings that include the optimization, diagnostic, code generation and solver settings in the models are safety compliant. Figure 44 is the screenshot of the results of running IEC 61508 modeling standards checks in Model Advisor for the Local Sequencer component in SymPLe architecture. Few examples of static checks on the model for IEC 61508 compliance are described below.

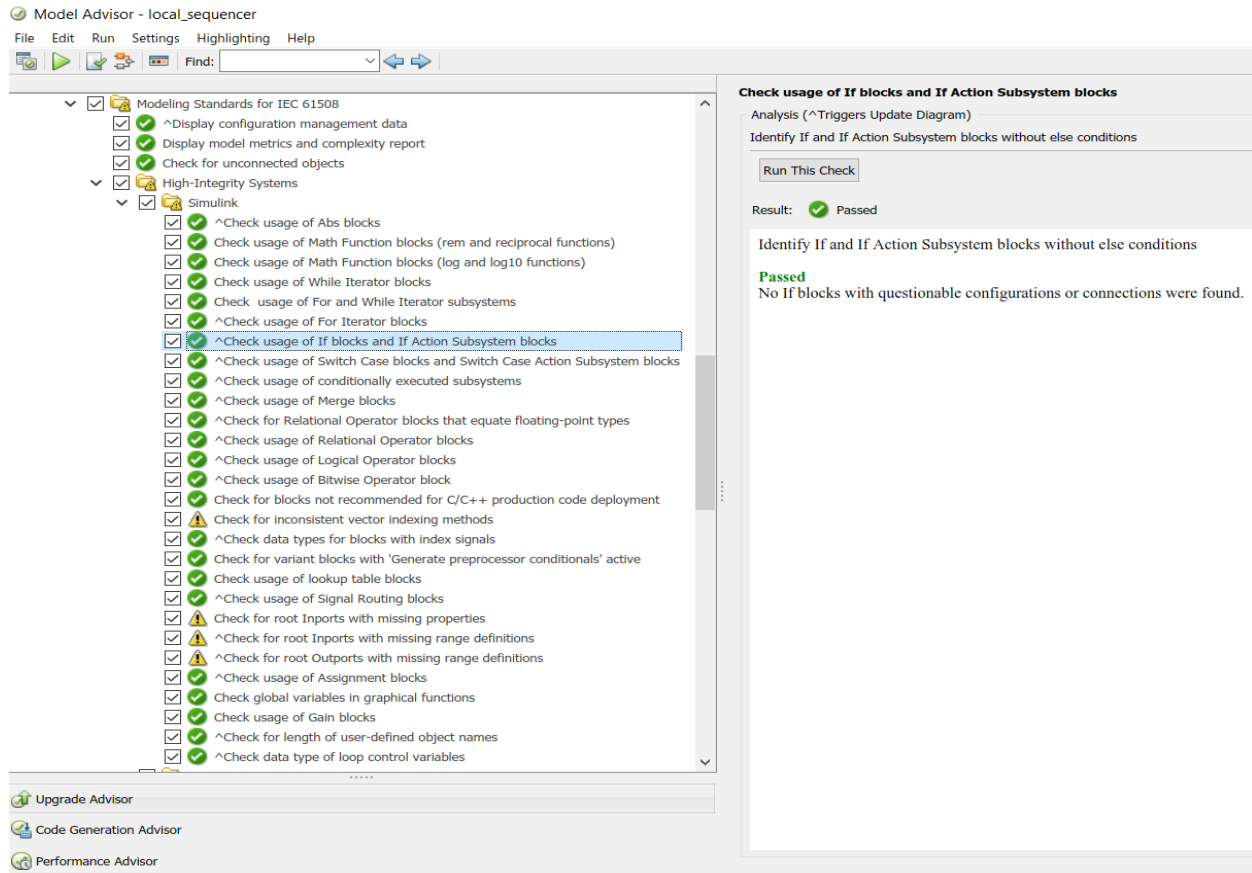


Figure 44: Static analysis checks for IEC 61508 compliance

Types of IEC 61508 Checks

This section presents few examples of the checks/rules of modeling standards for ensuring IEC 61508 compliance, available in Simulink Model advisor. The rule ‘Usage of If blocks and If Action subsystem blocks’ checks whether an ‘else’ condition is specified for an If-Else construct, to ensure that there is always a defined default action to be taken if the ‘if-else if’ conditions does not satisfy. Figure 44 shows that this check passes for the Local Sequencer component in SymPLe architecture. A rule on ‘Usage of Bitwise operator block’ ensures that signed integer inputs are not used for Bitwise operations as it might result in unexpected results due to the sign bit. Another check on ‘Usage of Logical Operators’ ensures that non-Boolean data types are not used for logical operations as it might end up in unexpected results.

Two of the checks that failed with warning (can be seen in the above Figure 44) are checks for root inports and outports with missing range definitions. This checks for input and output ports for which the minimum and maximum values that they can take are not provided in the properties. These minimum and

maximum values if provided, can be used for Design error detection by Simulink Design verifier to statically detect minimum and maximum value violations.

Model Advisor is also used for statically checking for HDL Code Compatibility. There are few blocks in Simulink library and few modeling constructs that are not supported for HDL Code generation. For example, continuous time blocks are not supported for HDL code generation. HDL code is hardware description language for digital circuits and thus only discrete time blocks are supported for HDL code generation. An example of a modeling construct that is not supported for HDL code generation is 'explicit type casting of variables'.

7.2 Design Error Detection

The Simulink Design verifier has the capability of finding certain design errors in the model without executing the model or feeding input values to the model. Static verification is sound which means that it covers the entire state and input space and so if a bug exists in the code, static analysis is guaranteed to report it. On the other side, static verification cannot be claimed to be complete, as it might result in false positives. Whereas dynamic verification is complete as it reports only issues that it has actually seen happening when feeding input vectors. Below given are the major design errors detected by Simulink design verifier during static analysis

- **Dead Logic** – Dead logic is detected if there is a portion of the model that is dead or never gets activated.
- **Divide by Zero** – Divide by zero error is detected if there is possibility that a divisor in division operation can be zero and there is no check in the design to prevent division by zero operation.
- **Integer Overflow** – Integer overflow design error is detected when there is possibility that an integer variable can be assigned value that overflows the allocated datatype size.
- **Out of Bound Array Access** – Out of Bound array access is detected if there is possibility than an array is accessed beyond its maximum number of elements eg: when there is no check to ensure that an array index lies within the maximum number of elements, before the array is accessed.
- **Non-finite and NaN Floating-point Values** – This design error is detected if there is possibility that a variable could take Infinity or 'Not a number' values. Not a number values are imaginary values, for example resulting from square root of negative numbers.

- Specified Minimum and Maximum Value Violations – Minimum and Maximum values can be provided for signals in the signal properties. If Design verifier detects possibility that the signals could take values outside the given valid range, it throws an error.

In addition to the design error detection and safety standards checks, static analysis also computes model complexity metrics and checks for requirements consistency.

7.2.1 Example: Dead Logic Detection

Figure 45 given below shows an instance of Dead Logic detected in Local Sequencer model. The block HDL Counter from HDL Coder library is highlighted for dead logic. This counter is a masked subsystem made up of other elementary Simulink blocks. This block has block parameters that allow the counter to be set in various modes like ‘Free Running’ or ‘Count Limited’ and fields to enter the Step value, Initial value and word length. The configured settings result in several logic inside these Simulink library blocks to remain unexecuted. For example, in local sequencer, this counter is used as free running counter with a step value of 1. Since it is configured as Free running count-up counter (due to positive step value), the paths in the subsystem that are for Count Limited mode and for count-down mode (negative step value) appear as dead logic and gets caught by the Model Advisor during static analysis. Such dead logic findings occurring as a result of configured settings of masked blocks are justified with appropriate comments.

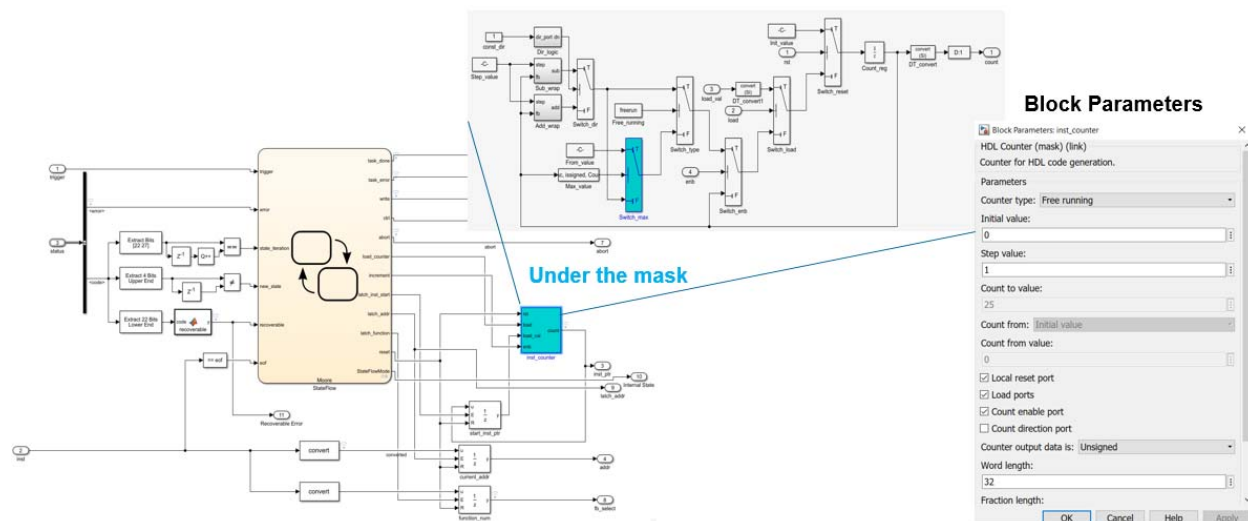


Figure 45: Dead Logic Detected during static Design error detection on Models

Chapter 8 Findings on Model-based Design Assurance

This chapter conveys the findings and lessons learned from the application of the comprehensive MBD&A on the representative safety critical system, the SymPLe architecture.

8.1 Design Flaws detected during Model based V&V

The below given Table 4 lists down all the design flaws found in SymPLe architecture during the V&V verification workflow. The important point to be noted here is that all the design flaws were found at model level, before the HDL code was even generated. The table provides the short description of the design faults that have been detected during the design assurance process. The field ‘Module’ gives the component in the architecture in which the design fault was detected. The identified faults were classified based on the perceived severity into low, medium and high impact categories. The severity of a design flaw is identified based on how the fault would impact the SymPLe system behavior if the fault were to manifest in an operational setting. Fault introduction phase gives the phase in the development life cycle at which the fault was introduced in the system. Finally, the Fault Detection V&V Method gives the technique/process step in the verification workflow in which the design fault was first detected. The design assurance process found faults at different stages of model verification: model testing, static verification, model coverage analysis, and formal verification.

Table 4: Faults Found during the presented Model-based Design Assurance Process

| Issue ID | Module | Design Fault Description | Severity | Fault Introduction Phase | Fault Detection V&V method |
|----------|-----------------|--|----------|--------------------------|----------------------------|
| 1 | Local Sequencer | Task Error is reset to 0 while still in Non-Recoverable Error state, whereas it is expected to stay high until the Global sequencer acknowledges the error by pulling the trigger signal low | High | Design/Model | Model-based Unit Testing |
| 2 | | Addr output is not latched to the fetch/write addresses | High | Design/Model | Model-based Unit Testing |
| 3 | | Inconsistency between Local Sequencer and Global Sequencer states could arise if error code changes from Recoverable to Non Recoverable after Local Sequencer | High | Design/Model | Property Proving |

| | | | | | |
|----|---------------|---|--------|--------------|--------------------------|
| | | enters its Error state. | | | |
| 4 | | Dead logic is detected in Inst_Counter block. This is due to the usage of HDL Counter with specific settings (Free running mode with positive step value) | Low | Design/Model | Static Verification |
| 5 | | Redundant condition '~recoverable' in state transition from Error to INIT state. | Low | Design/Model | Model Coverage |
| 6 | FB Controller | When ctrl signal goes high, FBController can move to Write_output state before the function block execution is actually completed ('executed' signal going high). The impact is that incomplete or incorrect results could be sent out of the system. | High | Design/Model | Model-based Unit Testing |
| 7 | | timeOutCounter will start only when the FBStateFlow goes to EXECUTE state. Therefore, if there is an error in READ_INPUT state and the FBStateFlow stays in READ_INPUT state due to a reoccurring error, there is no timeout error. | Medium | Design/Model | Model-based Unit Testing |
| 8 | | When the ctrl signal goes high, READ_INPUT state exits before all the inputs are read .When the ctrl signal goes high, WRITE_OUTPUT state exits before the outputs are written. | High | Design/Model | Property Proving |
| 9 | FB Functions | The order and mapping of signals in fb_control bus in FBFunctions does not match with FBcontroller. Enable Result signal in FBFunctions is mapped to latch output signal in FBController and Enable output signal in FBFunctions is mapped to execute signal in FBController, whereas it has to be mapped vice versa for the expected behavior. | High | Design/Model | Model-based Unit Testing |
| 10 | | While inputs are being read into the shift register, datatype error is signaled out well before all the inputs are fully read, due to the partial inputs. This error code be read in FBController while in Fetch state, causing the FBcontroller to go to error state. | Medium | Requirements | Model-based Unit Testing |
| 11 | | Divide by zero error code behavior is inverted. If there is no divide by zero error, the divide by zero Error code bit is set and if there is a divide by zero error, the error code is 0. | High | Design/Model | Model-based Unit Testing |
| 12 | | Overflow Error is not set when inputs - 2147483648 and -1 are fed as inputs to the | Medium | Design/Model | Model-based Unit Testing |

| | | | | | |
|----|-----------------|--|--------|--------------|---------------------------------|
| | | divide function block. | | | |
| 13 | | In Qmn type division, the dividend and divisor is interchanged. Value 2 is the dividend and value1 is the divisor whereas for integer type division Value 1 is the dividend and value2 is the divisor. | High | Design/Model | Model-based Unit Testing |
| 14 | | The Qmntodata conversion of the division result is faulty. Integer part is always appearing as 0 and fractional part does not correspond to the result. | High | Design/Model | Model-based Unit Testing |
| 15 | | ExecComplete signal is set to TRUE always, even if there are execution or data errors like divide by zero, datatype errors etc. or if there is a reset request from FB Controller. | High | Requirements | Model-based Unit Testing |
| 16 | | Function blocks pass on the input datatype as the output datatype even if it is invalid or incompatible with the function block. | Medium | Requirements | Model-based Unit Testing |
| 17 | | During underflow and overflow errors, the end results are not the saturated values. | Medium | Design/Model | Model-based Unit Testing |
| 18 | | Qmn type subtraction with negative differences, end up in results (both fractional and integer parts) that are different from the expected values. | High | Design/Model | Model-based Unit Testing |
| 19 | | Data type mismatch error in the logical functional blocks. Mismatch in bool to safebool conversion | High | Design/Model | Model-based Unit Testing |
| 20 | FB Controller | The Local sequencer is stuck in FETCH state as state change in FBController is delayed. | High | Design/Model | Model-based Integration Testing |
| 21 | Local Sequencer | The Local sequencer is stuck in WRITE state as it misses the state change in FBController due to an extra substate in Local Sequencer WRITE state. | High | Design/Model | Model-based Integration Testing |

8.2 Lessons Learnt during the Model-based Design Assurance workflow

Listed below are some of the key findings and lessons learnt during the application of the presented V&V workflow to the target system – SymPLe architecture.

Effectiveness of MBDA – Model Based Design Assurance has helped us in identifying several design flaws in the system. Figure 46 is a graph that shows the percentage of total design issues found in the SymPLe architecture during each verification phase. 21 design faults were found and remediated during the design assurance process. The design faults were classified into low impact (shown in grey in Figure 46), medium impact (shown in blue in Figure 46), and high impact (shown in orange in Figure 46) faults based on the severity of the system failures caused by them. About 81% of the total count of design issues were found during Model testing that involves all levels; unit, integration and system testing and 9.5% were found during formal verification at model level. Around 66.7% of the total design faults uncovered were high severity faults and all of them were found in Model Testing and Formal verification at the model level. Around 23.8% were medium severity faults, all of which were found in Model Testing phase. Around 9.5% of the total number of design issues were low severity faults and were typically found during static verification and model coverage analysis. Most importantly, all the design issues were found at the model level before code generation.

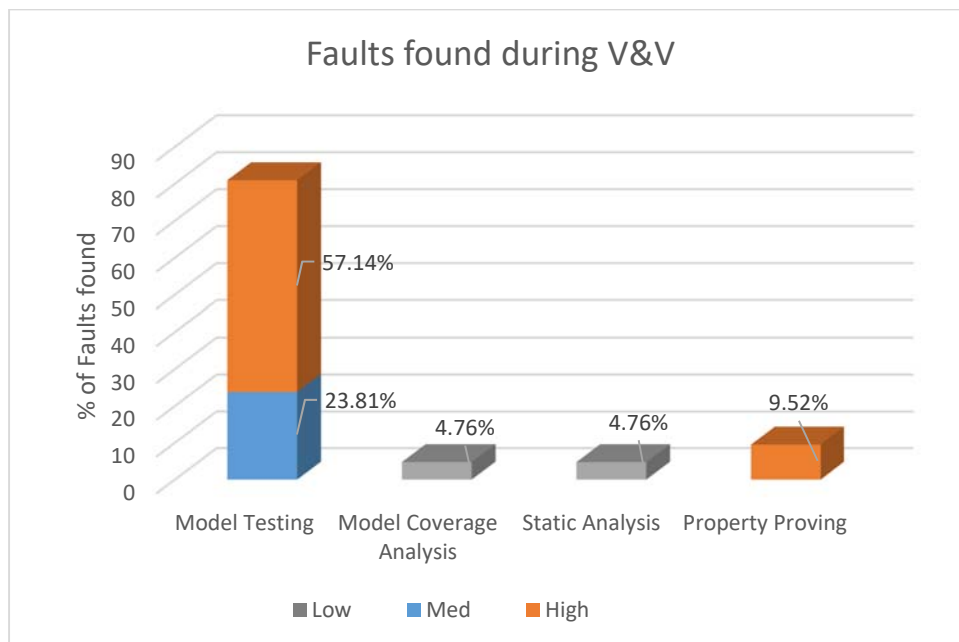


Figure 46: Design faults found in SymPLe architecture by V&V process

Model Testing as base for subsequent V&V phases - Model testing was found to be the most efficient and informative phase in the entire verification process. As model testing deals with dynamic execution it

helps to better understand the behavior of the system. The experience gained in this phase helped in guiding the subsequent verification processes. As shown in the graph in Figure 46, most of the design faults, about 81%, were identified in the model testing stage. There are very few faults that bypassed the Model Testing phase to be later on caught by the subsequent V&V phases. Model testing stage involved substantial manual effort in studying requirements and design and carefully formulating testcases to cover the entire model. The testcases developed at the model level acted as a set of base test vectors that could be reused for testing the HDL code and FPGA implementation of the system.

Synergy between model testing and formal verification – This exercise demonstrated an interesting synergy between Model Testing and Formal verification. Formal verification when conducted independently was not very successful as the proofs were not adequately informed on the behaviors we wished to prove. However, formal verification guided by model testing was found to be highly effective in finding ‘hard to detect’ design flaws. Model testing provided insights into where to concentrate and how to construct proofs for formally verifying critical parts of the architecture. Failed tests during model testing helped to identify vulnerable/weak areas in design that could be a potential source for software common cause failures (SCCFs). These critical areas in the design could then be targeted for rigorous formal verification.

The non-linear workflow model – ‘Vee’ diagrams are software development life cycle (SDLC) models that represent all the processes in a sequential V shape manner. The model-based development and V&V workflow prepared for the SymPLe architecture was adopted from the V model illustration in the IEC 61508 standard. The standard Vee model depicts an ideal linear workflow model. During the implementation of the design and assurance process, it was noted that the process did not follow the ideal linear path. The actual process revealed that each stage in the V model representation is incremental and achieves maturity over time through an iterative feedback process. Most of the time, the Requirement analysis process resulted in a set of feedback suggestions and queries on imprecise, ambiguous and contradictory requirements, as shown in Figure 47. These queries get addressed by the designers by clarifying and refining the requirements. The requirements also gets revisited when false test failures or low coverage cases are ascribed to wrong or missing requirements respectively. While false failed test cases informed us about missing/incorrect requirements and lead to revising the requirements, valid test failures were usually dedicated to design issues. These design issues identified during property proving and model testing were iteratively fixed in the model. Few other design errors like dead logic and over specified design are identified with low model coverage. Figure 47 shows the incremental development of the requirements, design and test vectors based on closed loop feedbacks from the subsequent verification

processes. This non-linear iterative workflow process was purely unplanned and evolved naturally as a result of the integrated nature of the model-based design and verification environment and the tool support.

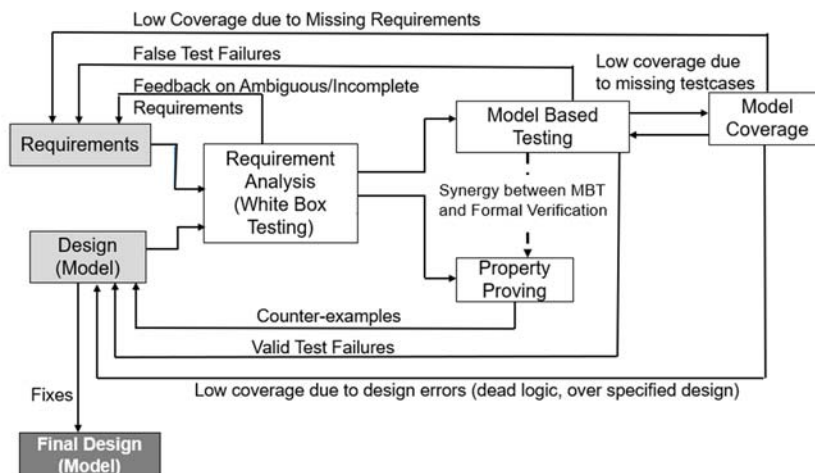


Figure 47: Non Linear Model based verification process

Importance of Hierarchical Model testing – This work shows the importance of progressively conducting the different levels of model testing. Even though unit testing was capable of finding several design flaws within the individual components, it was observed that few design issues escaped the unit testing phase. These design issues that were related to interactions between components could only be uncovered during integration testing. Thus, completeness to model testing is only achieved when all the stages in the testing hierarchy Unit, Integration and System testing are followed sequentially in order.

The Value of Traceability – Bidirectional traceability between requirements and model help the testers to correlate design elements to the implemented requirements, thereby gaining a good comprehension of the system design. This helps the testers to become very effective during white box testing. The traceability between model/requirements and code also guides the testers and reviewers during code level verification and code reviews. Traceability between requirements and verification artifacts (tests/proofs) helps ensure requirements coverage, completeness of verification and support IEC 61508 compliance.

Chapter 9 Model Based Fault Injection

9.1 Introduction

This chapter presents the last contribution of my research. In this chapter we introduce and develop a novel technique to model-based fault injection called *property based strategic fault injection*. This chapter first presents a basic introduction to fault injection concepts, and then a more detailed discourse on the benefits of property based fault injection over other methods to establish a base of understanding for the research, results, and contributions that are presented.

9.2 Background

With technology advancements and improvements in semiconductor electronics, we have been witnessing continuing decrease in semiconductor manufacturing process which is expected to reach 5nm process in 2020. The reduced transistor sizes, increased chip density, low voltage and high frequency operation make the electronic systems more prone to single and multi-event upsets, which can lead to system failures. Furthermore, critical infrastructure systems and CPSs in general are becoming software intensive systems where almost all aspects of the functionality of the system are defined by software constructs and architecture. Examples of these types of systems are autonomous systems, process control systems, medical therapy devices, power distribution, and smart transportation. The common thread among these types of systems is dependability.

Dependability evaluation involves the study of failures and errors and their potential impact on system attributes such as reliability, safety and security. It has been observed that identification of root causes of system failures in operational environment is very hard due to the disastrous nature of system crash or failure and large error latency. Thus, it is particularly difficult to reproduce a failure scenario for large complex systems just from failure logs and observations alone. The use of experiment-based or measurement-based approaches for studying Cyber Physical Systems dependability is gaining acceptance in many industries to better understand the effects of system faults, errors and failures on overall system safety and reliability[61] [62] [63] [64] [65] [66] [66] [67] [68] [69]. This is evidenced by the fact that functional safety standards like IEC 61508 and ISO 26262 identify fault injection testing as a highly recommended or mandatory validation technique for all safety integrity levels. Further, the overall safety and reliability of critical CPSs strongly depends on the fault tolerance and mitigation strategies employed,

and the correct implementation of these strategies; therefore, methods to credibly quantify and characterize the safety and reliability of CPSs is of great interest to the safety community at this time.

Fault Injection is defined as a dependability evaluation technique based on the realization of formal controlled validation experiments in which system behavior is observed while faults are explicitly induced in the system by the deliberate introduction (injection) of faults [70] [71]. That is, faults are injected into the system and the resulting behavior is observed. This technique speeds up the occurrence and the propagation of faults in a system for the purpose of observing the effects of faults on the system performance and behavior. Depending on the objectives of fault injection, the measures or the information from fault injection often serve different purposes.

With the recent popularity of model based design and development (MBDE) and testing, fault injection testing at the functional model level is gaining significant interest. The reason for this interest is it aids in detecting design errors and incorrect requirements on fault detection and tolerance features, very early in the lifecycle development process. Moreover it supports analyzing the failure modes and effects of faults (FMEA) on different parts of the design at the model level during design time, thus saving valuable time and improving product quality.

The work presented in this chapter describes a model based fault injection framework in the Mathworks Simulink platform. The innovation of the work is that the technique is property based and applies formal model checking verification methods at the functional model level of design thereby guaranteeing a near-exhaustive state, input and fault space coverage. The framework ensures complete fault injection coverage by offering an automated integration of saboteurs throughout the model. This FI framework also provides a saboteur that emulates a diverse set of fault models reflecting the physical HW aspects of a system, such as permanent, transient and delay faults.

9.3 Classical Fault injection and Strategic Fault Injection

To gain appreciation of model based fault injection, we review classical fault injection. Classical fault injection is similar to “black box” or “grey box” testing where there is little observability with respect to internal execution behavior of the target systems. Accordingly, classical fault injection is often characterized by its statistical nature, meaning that statistical models are used to govern the fault injection experiment [72]. When defining a fault injection experiment from a classical perspective, the fault space (Figure 48) of a system is important consideration. Fault space is defined as multidimensional space and

depends on three main factors like type of fault, time of fault occurrence and location of fault. The type of fault can be further divided into factors like the duration of fault and fault value.

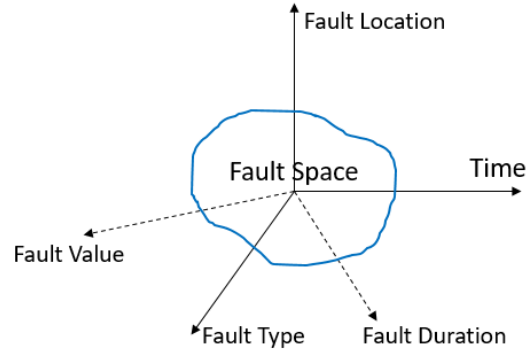


Figure 48: Fault Space

For fault injection analysis of a realistic system the experiment space includes other factors. For example, in the FARM fault injection model [73], F includes the fault space dimensionality of injection time, fault location, fault type, fault value and fault duration. A represents the activations or inputs to the system under test, R represents the readouts or measurements as fault injection is applied to the system, and finally M is the metrics to assess the dependability of the system from the collected readout data. If we expand F , all seven parameters that define a fault injection experiment are:

S_0 = Set of initial conditions

a = the set of external inputs

Δ = is the duration of the injected fault

t = fault occurrence time, or when the fault is injected into the system

l = fault location

f_m = a specific fault type as sampled from fault classes

v = Fault value

Note that the fault injection parameter space is multivariate. Considering the different fault model types (permanent, intermittent, transient, double faults, sequential faults, etc.), where and when we inject fault the number of permutations and experiments is virtually infinite.

A good overview on classical fault injection is found in [74]. With classical fault injection, the sampling process can be random or guided by some a-priori knowledge of the fault distribution. In all cases, an extremely important parameter in the design and assessment of safety critical systems is fault coverage, *C*. Fault Coverage is a metric; fault coverage denotes the conditional probability that the system detects or corrects a fault given that a fault is present in the system and is represented by the below equation [75] .

$$C = P(\text{Proper Handling of Fault}) | (\text{Occurrence of a fault} \in \mathfrak{z}), \quad (1)$$

where \mathfrak{z} represents fault space.

For a given fault injection experiment, let t_{pi} denote the instant of assertion of predicate p for experiment i , $i=1. \dots n$, let $Y_i(t)$ denote the random variable defined by,

$$Y_i(t) = I(t_{pi}) = \begin{cases} 1; & \text{if the assertion of } p \text{ is observed } [0, t] \\ 0; & \text{otherwise} \end{cases} \quad (2)$$

The random event described by the predicate p can be associated to a binary random variable Y , which assumes the value 1 when the predicate is true and 0 when the predicate is false during the observation interval T . The variable Y is then distributed like a Bernoulli distribution with parameter C . From the definition of the Bernoulli variable the parameter of the distribution equals the mean of the variable. In this case,

$$\begin{aligned} C = E[Y] &= 1. \text{Prob}(Y = 1) + 0. \text{Prob}(Y = 0) = \text{Prob}(Y = 1) \\ &= \sum_{f \in F} \text{Prob}(Y = 1 | F = f) \text{Prob}(F = f) \end{aligned} \quad (3)$$

The last expression is obtained by applying the theorem of total probability. It is generally assumed that the fault tolerance mechanism will always either cover or not cover a certain fault f , the probability $\text{Prob}(Y = 1 | F = f)$ is either 0 or 1, and can be expressed as a function of the fault,

$$y(f) = \text{Prob}(Y=1 | F=f) = \begin{cases} 1, & \text{if } f \text{ is covered} \\ 0, & \text{if } f \text{ is not covered} \end{cases} \quad (4)$$

From equation above, the following expression for fault coverage with respect to statistical fault injection experiment is obtained,

$$C = \sum_{f \in F} y(f) \text{Prob}(F = f) \quad (5)$$

The intuitive definition of fault coverage is that it is a measure of a system's ability to perform fault detection and fault recovery given the existence of a fault. Coverage is usually related to the fault detection and fault tolerance features of a system and the management of those capabilities to detect and mitigate those faults.

From a practical point of view, safety critical systems employ extensive fault detection/tolerance features to ensure proper fail operational and fail safe behavior in the event of faults. For example, FDIM (Fault Detection, Isolation, and Mitigation) software of the Teleperm TSX Reactor Protection System studied in [25], account for as much as 50 percent of the executable system software code. This code is rarely exercised in the real world because faults and failures are an infrequent occurrence. This FDIM code is vital toward system dependability and safety compliance, and can only be effectively tested and validated by realistic fault injection campaigns.

There are number of challenges and problems associated with classical fault injection. Namely, the most significant challenge is managing the combinatorial state space explosion problem due to the multi-variable nature of fault injection experiment. Several techniques have been published to help manage the state space problems which are called pre-injection analysis methods [76]. Most of these methods attempts to analyze the system before fault injection to; (1) improve efficiency by reducing or eliminating no-response faults, (2) eliminate faults where a priori knowledge is available on their detection performance, and (3) eliminate locations where faults have no-effect (e.g. unused memory and registers). Various advanced statistical experiment models have been proposed over the years to help manage the sampling of the multivariate space [77]. Using these pre-injection methods and advanced statistical methods allows one to explore the fault injection space more intelligently, concentrating on specific functions, locations, and times. Nonetheless, to demonstrate high levels of “fault tolerance coverage” (e.g. 99.999999%), the number of fault injections required is commensurate with the fault coverage requirement – which leads to very large number of fault injection experiments. As rule of thumb, the number of experiments needed for a given confidence interval and coverage level is approximately 10^{x+1}

where x is the number of “9’s” in the coverage factor. For example, for $C = .999$ at least 10,000 fault injection experiments are needed.

Another issue with statistical fault injection is the efficiency of testing in the presence of no-response faults. No-response faults occur when the system response to the fault does not indicate an error. If 50% of the fault injection experiments result in no responses, then twice the number of fault injection experiments are needed to achieve a given level of fault coverage. One of the purposes of pre-injection analysis is to increase efficiency by reducing no-effect faults; however, it has been shown that it is challenging to reduce no-effect faults to low levels (less than 10%). These problems are not unique to fault injection; it’s essentially the same combinatorial testing problem for complex digital systems.

Fault injector practitioners almost always have some a-priori knowledge or model of the faults (especially of the type of fault), but even then, the capacity of statistical random fault injection to find uncovered faults affecting safety is daunting.

9.4 Strategic Fault Injection

In recent years, the challenges with respect to statistical fault injection has stimulated interest in strategic fault injection methods - as a means to be more intelligent in the application of fault injection to a system. Strategic fault injection focuses on finding “high value or impact faults” that may lead to safety or security violations or hazardous conditions in a system. As such, strategic fault injection is not a statistical approach, but rather a search based approach. It aims to provide high coverage against specific unsafe system states that most likely result in safety violations or hazards. Examples of recent work include: Smith and Johnson in [78] describe a malicious fault list generation method to identify and inject “malicious faults” into railway safety controller. Alemzedah in [79] describes a STPA approach to fault injection for identifying and targeting faults that cause unsafe controller actions in a medical robot. Zonouz et al describe in [80] techniques for analyzing binary code to identify critical regions that effect safety and targeting those regions for fault injection. Jha et al use Machine Learning to maximally identify autonomous vehicle system behaviors that can lead to unsafe conditions [81]. The shortcomings to strategic fault injection are that it focuses on a small sub-space of the fault space. Unless all points in the input, state and fault space are sampled, there exists a possibility that a fault intolerant behavior of the system goes undetected during the verification. Our approach to adding value to the strategic fault injection research community is based on property based fault injection.

9.5 Introduction to model checking

Property based fault injection relies on a formal verification technique called Model Checking. Accordingly, we introduce at high level the concepts of model checking. Model checking is a formal verification technique that mathematically verifies the validity of critical properties of a model of a system. Model checking (Figure 49) refers to a set of techniques that take a system model and a formal specification (eg. temporal logic specification) and automatically determines whether the model matches the specification or not.

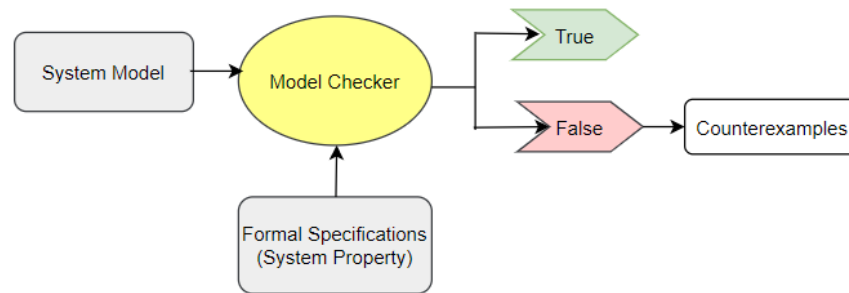


Figure 49: Model Checking

A “model” in model checking is a mathematical, logical representation of a system behavior. Here “system” means a behavioral representation of the SW or HW of the system. Models are usually expressed as non-deterministic automata or state machines, consisting of a finite set of states and a set of transitions called Kripke structures [82]. Model checking essentially performs a reachability analysis on the model under verification. With model checking, the user develops a “model” of the system using a model checking language. Examples, include Promela (SPIN tool), SMV_L(SMV tool), Simulink Function blocks (Design Verifier tool), and VHDL (Prop_check tool). The model represents the system to be verified as automata transition system. Transition systems are intermediate representations with the nodes representing reachable states and edges signifying the state transitions. Next, the user translates system requirements (Safety, Security, and Functional) into temporal logic formulas. These formulas are “checked” against the system model when model checking is conducted. If violation occurs, model checker returns a counter-example. If not, the model checker provides assurance that the “system” is compliant with respect to the requirements.

The basic model check problem is:

Let M be a state-transition system.

Let f be the specification in temporal logic.

Find all states s of M such that $M, s \models f$.

The Simulink Design Verifier tool we used in this research is a special model checker tool that allows users to graphically build the formal specifications to be checked on the Simulink system model using Simulink library function blocks. The Design Verifier tool works on the principles of Bounded Model Checking and K-induction techniques for solving the state reachability problem and satisfiability (SAT Solver) to check if the mathematical representations of the properties of a model are satisfiable [83]. In Design Verifier, the transition systems are denoted using

the triplet (S, S_0, T) , where S = set of all reachable states, $S_0 \subseteq S$ is the set of all possible initial states and $T \subseteq S \times S$ is the set of transition relations.

A property P denoted using a temporal logic formula is proven to be satisfiable if the property satisfies in all states reachable from the initial states.

$\text{Reachable}_T(S_0) \subseteq P$, where $\text{Reachable}_T(S_0)$ denotes the states reachable from S_0 using the transition relation T .

The reachability problem for a transition system (S, S_0, T) and a given property P is formalized as follows [83]:

$$\forall n \geq 0 : \forall s_0, \dots, s_n : \text{path}(s_0, \dots, s_n) \wedge S_0(s_0) \rightarrow P(s_n), \quad (6)$$

where $S_0(s_0) \leftrightarrow s_0 \in S_0$

The reachability problem is reduced to showing that every state reachable from the initial state s_0 satisfies property P . In this manner, counterexamples of size n are detected. The induction rule is used afterwards and generalized to show that if P holds in every state of every execution of length n , then every successor state $(n+1, n+2, \dots, n+k)$ also satisfies P .

There are several model checking tools that operate on specific modeling languages like C, Promela, Timed automata, SMV and so on, thus demanding the conversion of the design into a formalism accepted

by the model checking tool. Although testing can expose many design flaws, it is impossible to test a system exhaustively by covering all combinations of states, inputs and timing sequences. Model checking on the other hand is a verification procedure that involves an *exhaustive search* of the state and input space of the design to ensure that a system always meets a specific property under expected conditions and that there is no undesirable behavior due to any design flaws. If a property fails, a counterexample is generated showing the input conditions that resulted in the property failure. In traditional testing, the verification quality depends on the input test vectors. Unlike traditional testing method where the input and expected output vectors have to be fed into the system, there is no need to feed in input vectors during formal verification.

For addressing the software common cause failures, many safety critical industries like the avionics, nuclear, and automotive industry have embraced formal verification techniques. Functional safety standards like IEC 61508, ISO 26262 and IEC 61513 also specify formal verification techniques as recommended technique for higher safety integrity levels (SIL Levels 3 and 4). Applying formal verification can reduce the amount and extent of module and integration testing required. The confidence gained on model checking is because of their sound mathematical basis and cost and time effectiveness involved in verification as compared to the traditional testing method. There are two main challenges faced by Model checking (1) state space explosion problem (2) measuring the correctness and coverage of the modeled properties.

9.5.1 Property Coverage: Do I have all of the Properties

Determining whether or not the properties completely and correctly describe all the system behaviors is a major challenge with model checking. Simulation based testing has several defined metrics to quantify the coverage of test vectors on the model/code, which include model/code coverage, functional coverage, FSM coverage and so on. While coverage of simulation testing involves measuring the activations during test executions, measuring the coverage of formal verification is a difficult task as the properties that are expressed in LTLs or finite automata visit all reachable parts of the design. Figure 50 indicates how missing properties (Properties 3 and 5 shown in red) during property specification can cause several points in the input-state space of the system to be missed during property proving.

In this work which deals with proving fault tolerant properties on the system model, assessing the model coverage of the properties is not considered to be in the scope of this work. There are several researchers working on devising techniques to obtain formal verification coverage. While many of the initial methods

for formal coverage measurement are based on mutations [84] [85], many alternative methods have been developed later on that include proof based techniques [86] [87]. One interesting work is the study by Chockler et al [88] which uses mutations in FSM and re-evaluating the specifications on the mutated FSMs using falsity coverage and trivial coverage. An aspect of the design is considered to be relevant if the mutated version falsifies a property or trivially holds a property.

From a pragmatic point of view, almost always the designers and specifiers of a safety critical system correctly capture the major safety and security properties of the system. Where there is concern is at the minor/lower level properties.

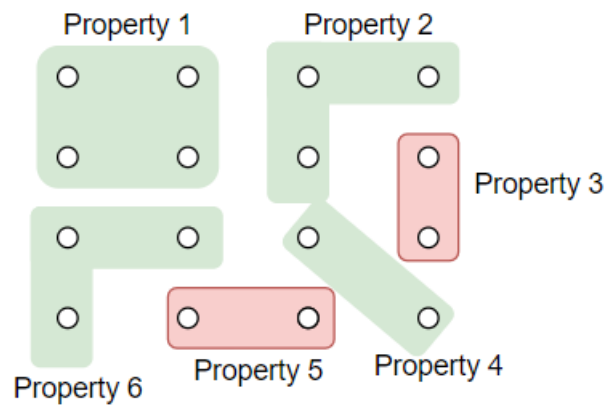


Figure 50: Uncovered state-output space points due to missing properties

9.6 Model Based Fault Injection using Property Proving

This work aims at studying the possibilities of performing an exhaustive fault injection at the model level using Simulink Design Verifier (SDV) property prover tool. As described in the related work section in Chapter 3, most of the works combining fault injection with formal verification are targeted for hardware design level. There are also several works that study the different model based fault injection methodologies. But this study is the first in its category to analyze the feasibility and effectiveness of formal verification for fault injection in a model based design environment. With MBDE offering a promising platform for many safety critical industries to conduct design and analysis, this study could be valuable for the safety critical systems practitioners to deal with problems of failure analysis, fault tolerance testing and fault coverage analysis at an *early design stage*.

Mathematically, the verification space of a system can be identified as the product of the input space and the state space of the system. Suppose a system has ‘ m ’ inputs each with ‘ i ’ different values and has ‘ n ’ state variables each of which can take ‘ s ’ different states. There are i^m points in the input space and s^n points in the state space. The exhaustive verification space involves feeding in all combinations of inputs for each combination of state variables. Hence theoretically, the total verification space is a product of the input and state space. V : verification space is calculated as:

$$i \in I, s \in S \rightarrow V = i^m \times s^n \quad (7)$$

where I – Total Input Space, S = Total State Space

For fault injection tests, there is also an additional parameter to be considered for the verification space calculation, the type of faults to be applied to the system. $V_f = I \times S \times F_{class}$, where V_f = Verification space during Fault injection, F_{class} = Fault space. For each combination of a sample point in the input space and state space, all applicable faults need to be considered.

$$i \in I, s \in S, f_n \in F_{class} \rightarrow V_f = I \times S \times F_{class} = i^m \times s^n \times f_1 \wedge i^m \times s^n \times f_2 \wedge \dots i^m \times s^n \times f_n, \quad (8)$$

where f_n is a fault type taken from fault classes

f_n represents specific fault models, like transient, permanent, SEU, etc.. The F_{class} fault space could be extremely large as it is a function of fault type, fault activation time, fault duration, fault location and the fault value.

As one can imagine, exhaustively covering this huge verification space using simulation tests by feeding in inputs, setting system to specific states and injecting faults one by one and verifying outputs is infeasible, if not impossible. This is one of the reasons statistical sampling is used in classical fault injection. On the other hand, if a designer could use formal model checking methods to perform fault injection, then fault injection could be nearly exhaustive with respect to the properties of the system. With property based fault injection the tester specifies the desired system behavior during a fault injection scenario and then the model checker tries to verify or disprove the fault tolerance behavior of the system. As property proving based fault injection is mathematically based, the state space, fault and input domain considered are over sets of sets which is potentially infinite. This mathematical completeness (with respect to a property) makes the technique efficient and effective as to not miss any point in the

input/state/fault space, which is a significant problem with traditional statistical fault injection. Thus the major advantage of the property proving based fault injection is its capability to verify the fault tolerant properties of a design exhaustively covering the entire state, input and fault space. The downside to property based fault injection using model checking is the state space explosion problem [89]. If the “system model” is significantly complex and large, then model checker may not be able to resolve reachability of all states in a feasible amount of time. Fortunately, over the past 15 years there have been great advances in alleviating the state space explosion problem in model checking. That said; care must be taken when performing model checking to avoid state space explosion issues. Methods like decomposing the system into smaller verification units are often used.

The basic idea of property proof based fault injection is shown in Figure 52. The first step is to specify the Fault tolerance properties and the Fault activation conditions. Fault tolerance property denotes critical system behavior or safety properties that needs to be met by the system outputs in the presence of faults. The Fault Tolerance Properties to be verified in the presence of faults need to be specified in a formal language. In Simulink, the Design Verifier language is based on a *Metric Temporal Logic* expressed graphically using several temporal operators, objectives and constraints blocks.

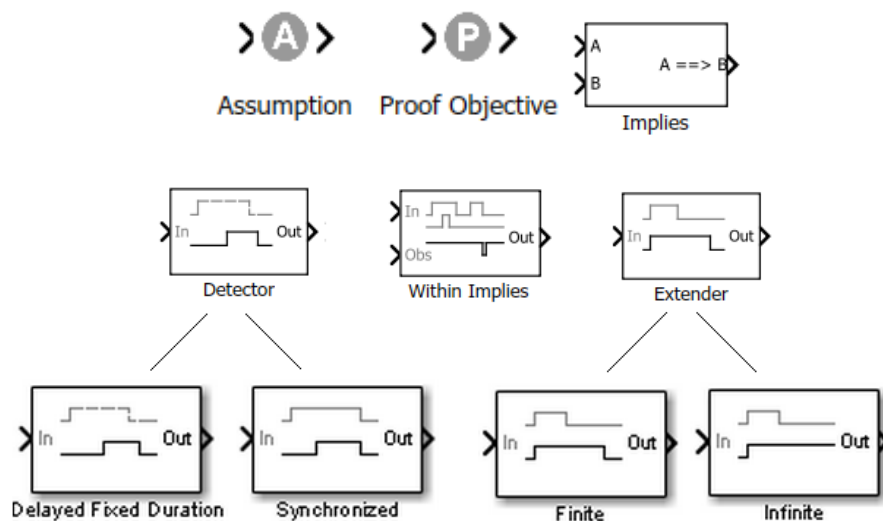


Figure 51: Simulink Design Verifier Blocks: Proof Objectives, Constraints and Temporal Operators [90]

Simulink DV library provides several blocks (shown in Figure 51) to specify the verification objectives and constraints during property proving [90].

- **Proof Objective:** Used to define objectives that signals must satisfy when proving model properties.[90] Separate values or range of values (closed or open interval) that a signal must achieve during proof simulation is provided in the ‘Values’ field in the block.
- **Proof Assumption:** Used to constrain signal values when proving model properties.[90] Separate values or range of values (closed or open interval) that a signal is assumed or permitted to hold during property proving is provided in the ‘Values’ field in the block.
- **Implies:** ($A \Rightarrow B$) Used to specify condition that produces a certain response.[90] This block is used commonly for representing property proofs, to state that the system should respond (response ‘B’) appropriately, with a particular input stimulus (stimulus ‘A’) i.e. receiving A as true should imply B is also true.

Simulink DV library also contains few temporal operator blocks that can be used to specify temporal behavior or timing aspects of the design in properties. They are given below [90]:

- **Within Implies:** (‘Within’ In) \Rightarrow Obs. Used to verify that a response occurs within desired duration.[90] This is a temporal variation of the ‘Implies’ block. It is used when a particular response (‘Obs’ input) from the system is expected to happen for atleast one time step within each true duration of the input stimulus (‘In’ input).
- **Detector:** Detects true duration on input and constructs output true duration based on output type [90]. The two different modes in which Detector block can operate are:
 - **Delayed Fixed Duration:** Once the input is detected to be true for specified time steps, the output is activated after a specified optional delay for a fixed specified time duration.
 - **Synchronized:** Once the input is detected to be true for specified time steps, the output is activated and stays true as long as the input is true. The output is synchronized with the input.
- **Extender:** Extends the true duration of input.[90] The Extender block operates in two different modes, a finite mode where it extends the true duration of the input signal by a user-provided number of steps and an infinite mode where it extends the true duration of the input signal indefinitely.

These Simulink DV library blocks, shown in Figure 51, are used to specify the properties to be validated. The proof objective block identifies the properties to be proven by Design Verifier. Injected faults might

be activated only in specific state of the system and with specific input values. Fault activation conditions specify the input/state sequences that causes the injected fault to manifest as error and propagate within the system design. To activate a fault, the necessary input/state preconditions, needs to be specified as constraints. Constraints are specified using the assumption block in the Design Verifier library.

To access the fault tolerance mechanisms of a safety-critical system basically requires two things; (1) the selection of a set of representative physical faults models must be accomplished, and (2) the application (e.g. injection) of those faults into system model. Item 1 constitutes the representative fault space of the system. Item 2 is the execution of the fault injection. For property based fault injection, under the given preconditions, the model-checker (Design verifier) systematically explores the state space, input space and fault model space with respect to a given property. The model checker automatically injects the faults during the state space search to find a violation of the fault tolerance property. If no violation is discovered, the property is satisfied, thereby ensuring that the system exhibits the safe behavior in the presence of any fault from the considered fault space. While maintaining the assumptions, if any violation of the property is encountered, Design verifier falsifies the property and generates a counterexample showing the input vector and fault vector that caused the safety property to fail.

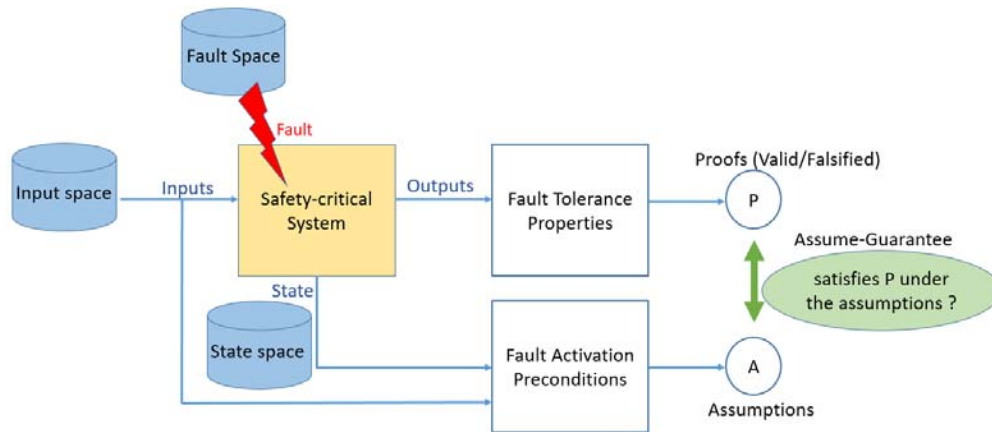


Figure 52: Fault Injection with Property Proving

9.6.1 Challenges with Property based fault injection

Several challenges in realizing the property-based FI at the model level were identified. They are:

Challenge 0: To find a way to incorporate faults as inputs into the system for model checker.

Challenge 1: To find an intelligent way to automatically identify the fault activation conditions.

Challenge 2: To design and implement saboteur.

Challenge 3: To realize automatic insertion of saboteurs in model.

9.6.2 Classical Fault Injection vs Property based fault injection.

Classical fault injection experiment covers a single point within the fault injection experiment space of the system, per experimental trial. A single point in the fault injection experiment space of a system can be considered to be one of the many possible combinations of points in the system's fault, input and state space. A trial is one execution of a fault injection process. A campaign is made of many trials where fault injection parameters are varied from trial to trial. As such, with classical fault injection there are always points in the FI experiment space (input/fault/state space points) that are not covered, which could overlook a faulty part of the design to go unnoticed. In contrast, fault injection based on property proving works at the property level. Given a safety property and a functional model, property based fault injection exhaustively searches the entire fault, input and state space for safety property violations, by covering all possible faults in all possible input and state conditions. So as shown in Figure 53, while classical fault injection checks for one experiment point at a time; formal verification based fault injection checks a set of points in the FI experiment space at a time (a set of experiment points constitute a property). We postulate with good reason that experiment points that are not covered with property based fault injection can be separately covered using randomized or strategic fault injection – as this same approach is used to augment SW combinatorial testing.



Figure 53: Classical Fault Injection vs Property Based Fault Injection

The classical fault injection approach at the model level (shown in Figure 54), involves continually conducting a set of simulation tests (called trials) to inject faults into the model and observing the

response of the system to collect evidence for estimating fault detection/tolerance coverage of the system. The process starts from the selection of faults to be injected, that includes the location, fault type, duration and injection time of fault. The saboteur for the selected fault model needs to be modeled and inserted in the selected fault location in the model. The fault injection inputs have to be inserted into the system to reflect the selected fault. These steps together define the ‘Fault’ attribute in the FARM model. There could be several input-state trajectories (often called the workload or input profile) that can activate the injected fault and propagate it until it reaches the fault detection/tolerance function. Out of the several input-state trajectories, one of them has to be selected for a test, and the input sequence is fed into the model based on that. After execution of the model with injected fault, the readouts are taken by using monitoring functions – such as scopes, display blocks or logic analyzers and analyzed to see if the fault was correctly detected and mitigated. All these steps have to be repeated for each individual fault injection trial – changing the input profile conditions and different sets of faults. A complete set of trials is called a fault campaign. Thus we can see that classical injection is a long process and the coverage of the fault, input and state space depends on how many FI tests were conducted or the number of FI trials.

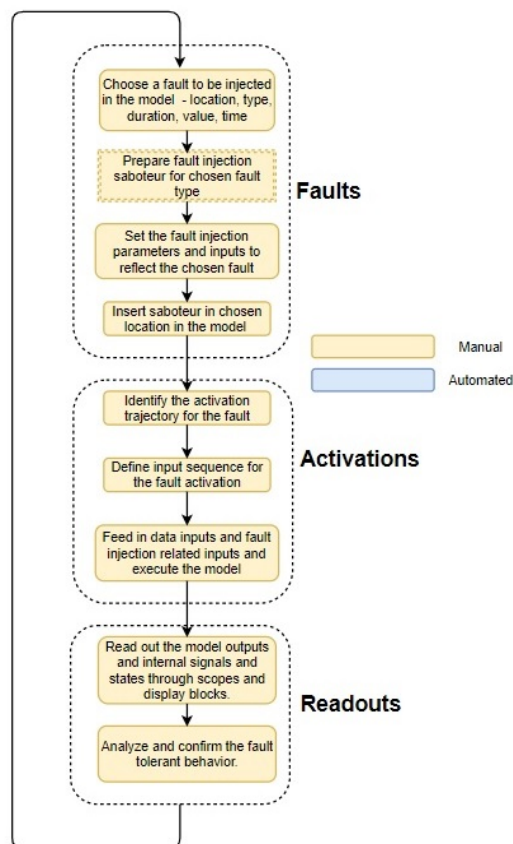


Figure 54: Classical Fault Injection Workflow

In contrast, Property based Fault injection process (shown in Figure 55) described in this thesis depicts a formal *search based process* that does not rely on statistical models, rather model checking and logic to accomplish its goals. The process is predominantly automated while still guaranteeing a comprehensive coverage of fault, input and state space. Below are the preparation steps to be performed before starting the property proving within Simulink DV.

Instrumentation of Model with saboteurs - The process starts with an automated insertion of a generic saboteur into all signals in the given test model. In the model-based context, a ‘*Saboteur*’ can be defined as a special module added between signal drivers and receivers in the model based design, which when activated alters the value or timing characteristics of the signals, thereby simulating faults for Fault Injection[91]. The saboteur in this FI framework is a generic one and supports multiple fault models. The saboteur instrumented model generated in this process is further used for the property based fault injection process.

Fault propagation/Error conditions modeled as assumptions - With automatic saboteur insertion we end up with a very large set of fault locations within the model. Thus manually finding the activation condition for each fault location could be an extremely time consuming process. It would require complete analysis of the model and identifying all possible input scenarios that can activate the faults – which is infeasible for system models commensurate of practical system scales. The approach we employ to solve this problem is to use the model checking tool to automatically identify the fault activation conditions (e.g. the input conditions) for any given fault. Fault activation condition denotes specific input conditions that can cause the injected fault to manifest as error and propagate the erroneous behavior to the boundaries of the component where the error can potentially violate a safety property. In this method, instead of manually exploring the entire input space to identify and specify the input conditions to activate the faults, the tester has to specify the state/output condition that indicates an error or system failure as an assumption that becomes true after a fault is injected. The error propagation condition is modeled using the assumption block in Simulink DV library.

Fault Tolerance Properties modeled for proving - Finally, the fault tolerance properties derived from the safety requirements of the system that needs to be validated are modeled using the proof objective blocks in Simulink Design Verifier library.

After completing all the above preparation steps, property proving is executed in Simulink Design Verifier as described below.

Choosing fault models from the representative fault space- After generating the saboteur inserted model and specifying the error conditions as assumptions and safety properties as proofs, we initiate property proving in Simulink Design verifier. In property proving mode, Simulink DV proves that the properties of the model satisfy the specified criteria. The properties are proven in the presence of different kinds of faults. The faults injected into the model, should be representative of real world faults. Thus, the appropriate fault models like Single Event Upset, Bit-flip, transient, permanent, Stuck-at-X, etc. are elaborated into the saboteur module. Fault injections are automatically carried out by the model checking tool. The DV model checker exhaustively covers all specified faults for proving the property.

Identifying fault activation conditions for each fault - With the erroneous/fault propagation behavior being specified as an assumption in the model, the Design Verifier explores the entire input space and identifies all the input sequences that can cause the injected fault to manifest as an error and propagate. In addition to relieving the tester from the laborious task of identifying the activation input sequences for each fault, it also ensures a complete fault activation space coverage (from within the input and state space) for each and every faults.

Checking for Violations of the safety property - Finally, on running the property proving on the specified properties with proof objectives, it either results in valid or falsified results. Validated properties would mean that the fault tolerance functionality is capable of tolerating the specified faults (within the fault space) that are injected. If the property falsifies, a counterexample is generated which shows the fault that bypassed the tolerance functionality or the safety feature. The location, duration, type and time of injection of the fault is clearly depicted in the counterexample. This allows for analysis of the inherent bugs within the design of the tolerance features in the model and to identify any uncovered faults that violate a safety property of the system.

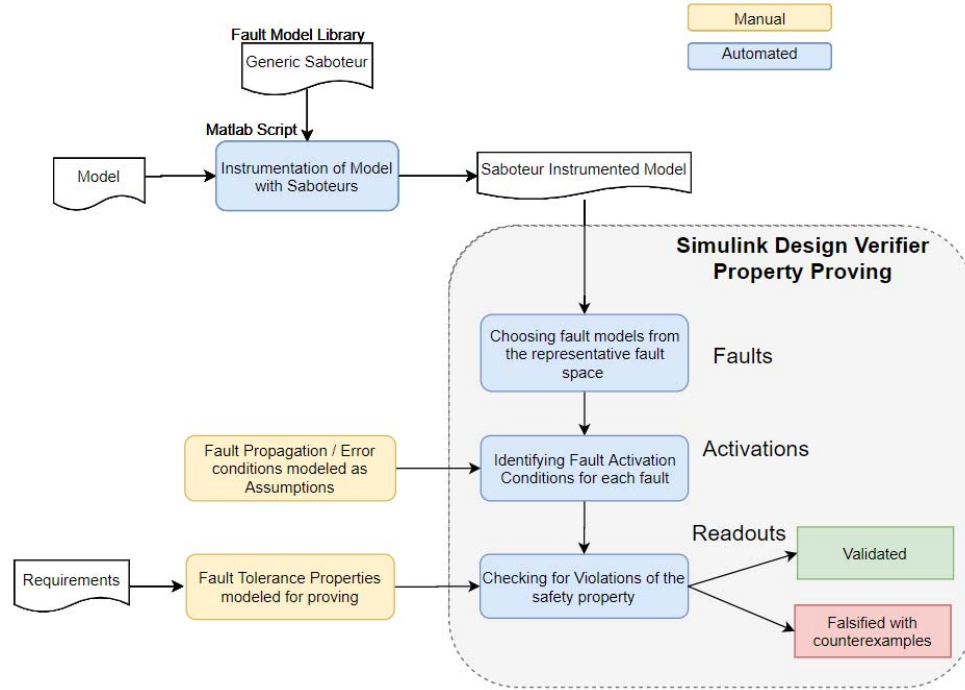


Figure 55: Property Based Fault Injection Workflow

9.6.3 Property Based fault injection - Fault-Input-State space Coverage

Figure 56 shows the effectiveness of the property based fault injection approach in terms of the coverage of the fault, input and state space. Referring to Figure 56, the left side enumerates the assumptions made by the model checking tool during model checking based on the constraints we specify in the model. These assumptions are associated with the capacity of the DV model checking to represent these parameters. The model checking tool assumes that the fault location control signal can take any value from the defined fault location IDs. This ensures that the Design verifier explores the entire fault location dimension within the fault space. There are no constraints placed on the ‘InjectFault’ boolean signal. So it allows model checking tool to assume that the fault can be injected at any time instant during an input/state sequence. This causes the model checker to completely explore the entire fault activation time dimension of the fault space to find property violation. Constraints are placed in the property specification to direct model checking tool to consider only the valid fault type values. Thus Design verifier explores and covers all fault types within the defined fault model list. Specifying no constraints for the fault duration signal causes design verifier to assume any value from the entire datatype range thereby allowing Design verifier to cover the complete Fault Duration space during model checking. With no constraints on the BitNum signal, the model checking tool is allowed to assume single or multi bit faults on any of the 32 bits in the signal thus causing the model checker to completely explore and cover the Fault Value

dimension of the fault space. Finally, another constraint that is specified, is to assume that a fault that is injected always activates, i.e. manifests as an error and causes an output malfunction or system failure. Thereby, the model checker exhaustively explores the entire input and state space and identifies all possible input and state sequences that can activate the considered faults in the fault space. In this manner, each fault tolerance or safety property verification, completely covers the corresponding fault activation space from within the entire input and state space. Thus, by combining all constraints on the fault injection control signals, and error propagation condition, this property based FI method is capable of covering all points along all dimensions in the fault space and all fault activation points in the Input and State space.

The challenge 0 identified for realizing Property Based FI is addressed with this FI framework by having the Fault Injection and Fault Model parameters as inputs for the Model checker to explore. Similarly the challenge 1 is also addressed by specifying the Fault propagation conditions as assumptions, thereby directing Model checker to automatically identify the fault activation conditions.

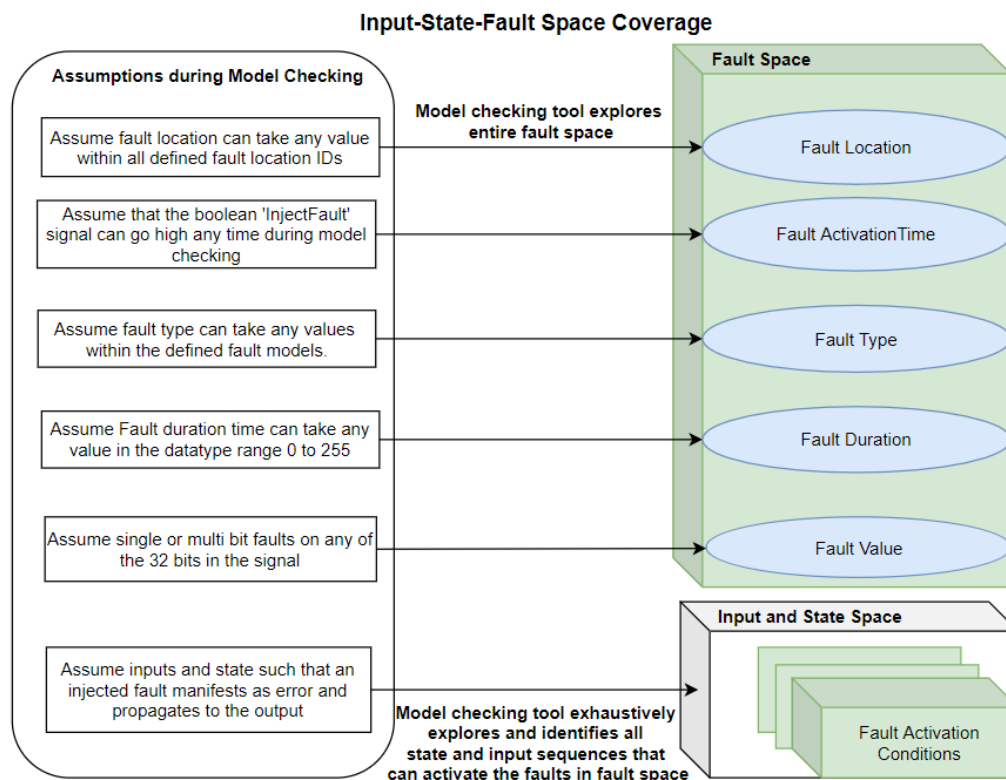


Figure 56: Property Based fault injection - Fault-Input-State space Coverage

9.7 Design and Implementation of a Comprehensive Fault Injection Framework in Simulink

9.7.1 Faults and Fault Model

A fault model describes the behavior of different faults that could occur in a circuit or target system. As the target system for our research is FPGA-based, the fault model developed in this work has been derived from the set of possible physical faults that could occur in an FPGA device and applied at the functional model level. This FI framework proposes and implements several physical fault models as shown in the Fault model taxonomy in Figure 57. This work focuses on the Transient and Permanent fault categories of physical faults. Note that Property Based Fault Injection is not limited by the fault model taxonomy shown in Figure 57, it can emulate a potentially large number of fault condition. For this early work, we have concentrated on fault models related to FPGAs. In an FPGA, soft errors can cause transient bit flips affecting the user-defined memory elements, flip-flops, and the combinational logic of the Configurable Logic Blocks (CLBs). The glitches in combinational logic can also be propagated to cause bit flip faults in sequential elements like flip flops and memory elements which will hold the incorrect value until rewritten with correct data bit. These transient single and multi-bit flips at each signal level are the first kind of faults considered in the fault model for this model-based FI framework. Gate delay and Path delay faults can cause delay in propagation of signals thereby making the device outputs to not meet their timing requirements and cause system failures or performance degradation in real-time high performance systems. This work also considers delay fault models to enable conducting timing related fault injection tests on the functional model of the target system.

Soft Errors causing bit flips in the configuration SRAM bits in an FPGA are irreversible until the application bit stream is re-flashed onto the device. Other causes of permanent faults in FPGA are manufacturing defects and aging related issues. These permanent faults contaminates the circuit design and thereby the function of the design. To simulate permanent faults at the signal level in functional models in Simulink, we have considered stuck at 0 and stuck at 1 faults in the fault model of our model-based FI framework. Single or multiple bits in a signal within the model can be stuck to 0 or 1 in this FI framework, thus simulating a permanently corrupted design. Another possibility of permanent faults considered are the ‘Stuck open or Stuck on transistors’ at the circuit level that can result in non-logical or ambiguous values on gate outputs. These faults are incorporated at the functional model level as ‘Floating’ signals that exhibit randomly varying 0 or 1 values. In this FI framework, the ‘floating’ fault model is applicable for single or multiple bits in a signal within the functional model.

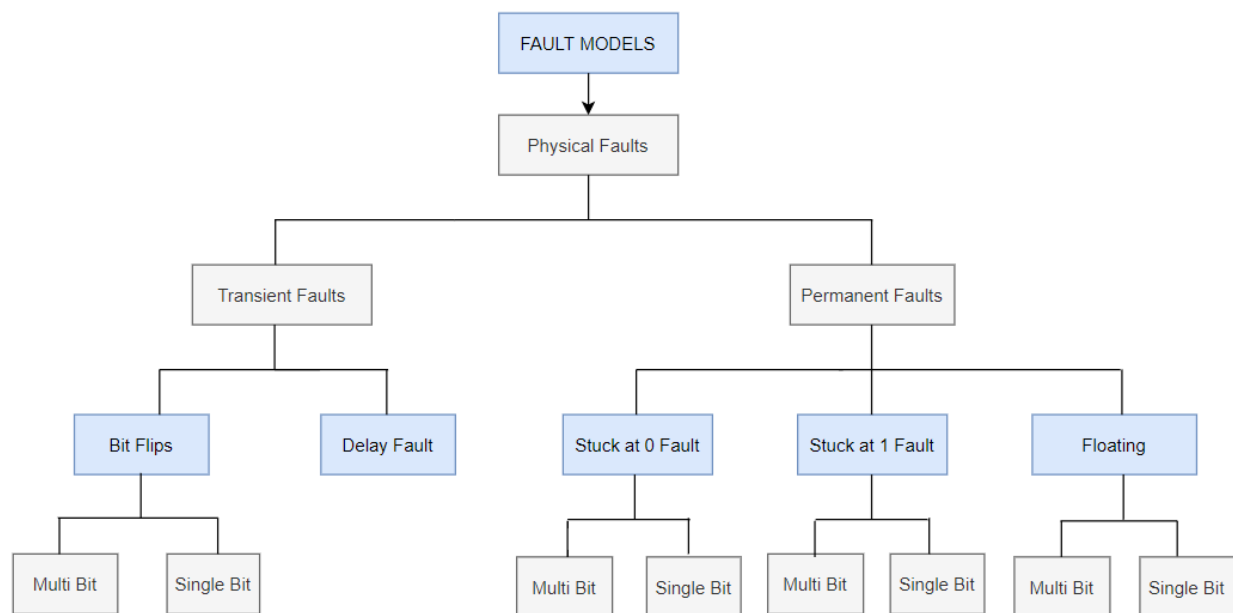


Figure 57: Fault Model Taxonomy

9.7.2 Saboteurs

The Saboteurs and Fault Injection framework developed in this work is a generic framework and is not tied to Property based fault injection. It can be used to facilitate different fault injection methods that include Property-based, Random, Strategic and Sequenced Fault injection techniques. This framework supports semi-automation of the FI process by enabling fault injection control within the functional model and automating saboteur insertion throughout the model.

The term *saboteurs* had been introduced in the context of fault injection on VHDL designs. *Saboteurs* are VHDL components that are added into VHDL code to alter the value and timing characteristics of a signal for injecting a fault [75]. The advantage of saboteur as compared to mutants³ and simulator command techniques is its capability to simulate a wide variety of faults including pulse, bit flips, delays, stuck at faults, open line and short [92]. This work uses saboteurs at the functional model level to achieve a diverse set of fault models for a given system design developed in Simulink. The generic fault model in the Saboteur encompasses faults typically encountered in FPGAs. It is discussed in section 9.7.1. As Simulink offers a highly flexible design platform with its rich set of toolboxes and libraries, it is possible

³ Mutation is a static fault injection technique in which slight alterations (like altering function operators, changing value assignments, implementing physical faults like stuck-at or a bit flip) are deliberately introduced in the model, HW or software design of a system to simulate different fault models. The resulting altered design is called a mutant.

to create multiple fault simulation blocks within a single saboteur subsystem. The challenge 2 identified and listed in section 9.6.1 is thus addressed with these Saboteurs implemented using Simulink basic blocks and encapsulated in a masked subsystem.

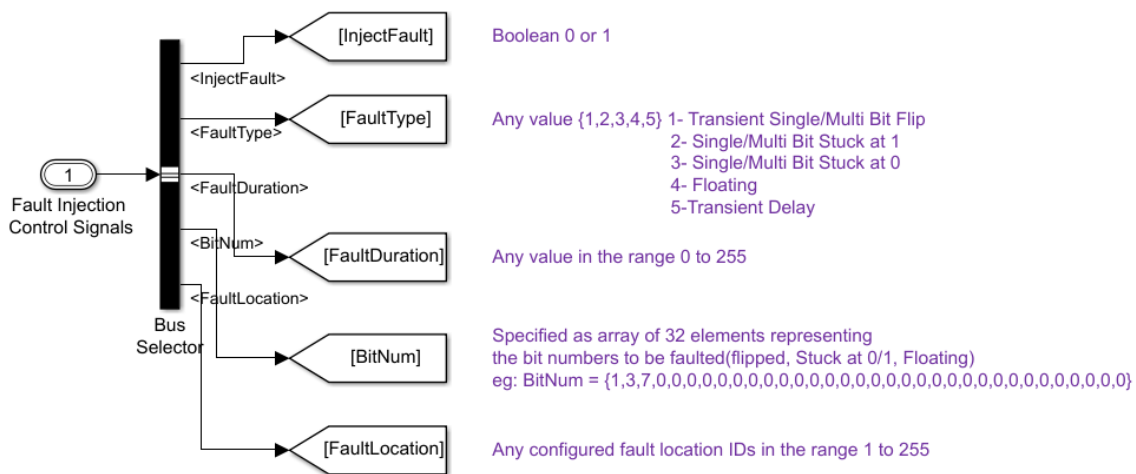


Figure 58: Fault Injection Control Signals

The control signals for fault injection shown in Figure 58 are InjectFault, FaultType, FaultDuration, BitNum and FaultLocation. The FaultLocation is a number identifier assigned to each saboteur inserted in the test model. The option to assign the fault location ID for a saboteur is provided in its Block parameter settings as shown in Figure 60. The functionality of a saboteur is hidden from the user by encapsulating the implementation within a masked subsystem (Figure 59) in Simulink. As shown in Figure 61, a saboteur's fault simulation is only activated if the fault location provided in the fault inject control signals match the Faultlocation ID assigned to that saboteur. The fault saboteur is implemented in such a manner that they just pass through the actual input in case the fault location control signal does not match the location ID assigned to it. If the fault location control signal matches the location ID assigned to the saboteur, the faulty output is fed out based on the type of fault requested in the control signals. FaultType is the control signal that is used to feed in the type of fault to be simulated within the generic saboteur. Each fault is assigned a different fault type value with '1' representing Transient Single/Multi Bit Flip, '2' denoting Single/Multi Bit Stuck at 1, '3' denoting Single/Multi Bit Stuck at 0, '4' denoting the permanent Single/Multi Bit Floating fault and '5' representing Transient Delay fault (shown in Figure 62).

'FaultDuration' control signal is only applicable to the Transient multi and single bit fault types and it specifies the duration for which the transient fault is expected to be active. It supports values in the range 0 to 255 which indicates that the transient fault (bit flips / delay fault) can be made to stay active for 0 to

255 clock cycles. 'InjectFault' is a boolean control signal that is set to true when a fault has to be injected in a signal. For transient faults, when this signal goes high, the faulty value obtained by flipping bits, or delaying the signal are send out for a duration equal to the 'FaultDuration' control signal. For permanent faults like Stuck at 1, Stuck at 0 or Floating signals, when the 'InjectFault' signal goes high the signal is set to the faulty value and stays permanently in that value. The 'BitNum' control parameter is used to specify the bits of the signal to be faulted (flipped, stuck at 0 or 1, floating). To specify multiple bits to be faulted, a 32 element array, with the bit numbers to be faulted specified in one-based indexing format, needs to be fed into the 'BitNum' control parameter. The 'BitNum' parameter has a dimension of 32 elements which indicates that up to 32 bits in a signal can be faulted (flipped/stuck at 0 or 1/floating) during fault injection. Eg: when this array {1,3,7,0} is fed into the 'BitNum' parameter with 'FaultType' selected as '1 – Bit Flip', it will result in transient bit flips on the 0th, 2nd and 6th bits of the signal.

Riesgo and Uceda in [93] have come up with fault model definitions at the RTL level. The faults on data and expressions as experienced at the RTL level in [93], are also achieved with the saboteur insertion within the Simulink model of the FPGA design.

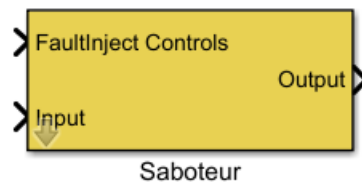


Figure 59: Saboteur Masked Subsystem

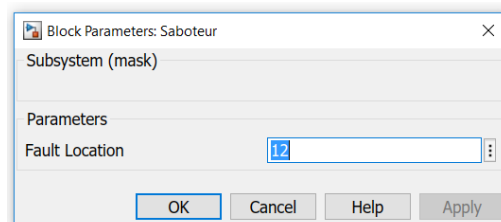


Figure 60: Saboteur Location Parameter

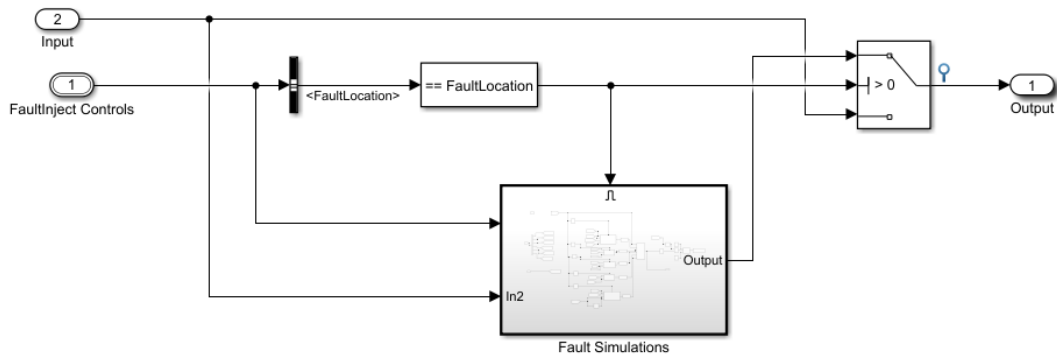


Figure 61: Inside the Saboteur subsystem: Level 1

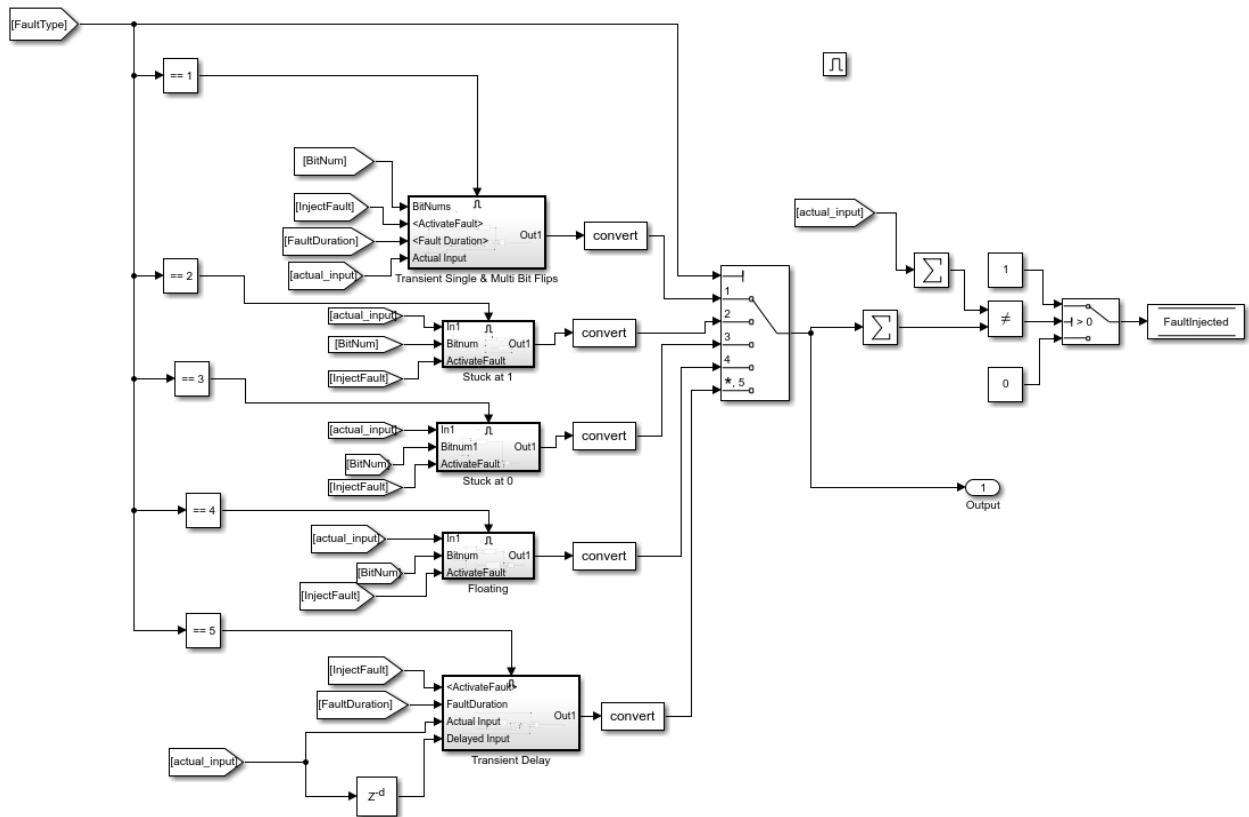


Figure 62: Inside the Saboteur subsystem: Level 2

9.7.3 Automated Saboteur insertion in model

Completeness of fault injection can be achieved only by covering the entire input, state and fault space of the model. Fault space coverage is dependent on the coverage of fault locations, fault types, fault injection time, fault duration and additional parameters like the bit number for the bit flip, stuck at X or floating faults. Faults occurring within a system at specific times can have adverse impact on the system outputs. Hence it is necessary to consider fault injection at all signal points available at the model abstraction level. A Matlab “m” script is developed to automate the insertion of the generic saboteur blocks within all signal lines in the model that is being subjected to Fault injection. Matlab commands are available that can search through the entire model and populate all blocks/ports/annotations/lines within a model. The command ‘find_system’ returns a vector of handles of all lines in the specified Simulink model. The search depth can be adjusted to include all lines within the top level system and its children, thereby ensuring complete signal coverage for design level fault injection. The lines are analyzed one by one and parameters like Source Block, Source port, Destination Block and Destination Port for each line are retrieved using Matlab commands. With these parameters in hand, each line in the Simulink model is deleted and saboteur subsystem is inserted between the source block and destination block. The saboteur insertion ‘m’ script runs fast, for example, takes about 5mins to insert 50 saboteurs within a single ADD function block. The automated saboteur insertion technique is limited to Simulink signals and cannot handle fault injection on stateflow registers, because at the model level the state holding register cannot be accessed for writing into. Figure 63 shows a model with saboteurs inserted in it. The algorithm of the saboteur insertion algorithm is given below:

Algorithm:

- *Copy the target system model to a new model for saboteur insertion.*
- *Load the bus datatype of the ‘FI control signals’ bus to Matlab workspace*
- *Read subsystems one by one from the user provided list of subsystems which needs to be subjected for FI.*
- *Within the subsystem, insert ‘Inport’ for FaultInject control signal Bus, connected to a ‘Goto’ port with ‘InjectFault’ tag.*
- *Retrieve the list of all the line handles for this model*
- *Parse through the list of all lines and extract the line properties ‘SrcBlock’, ‘DstBlock’, ‘SrcPortNumber’ and ‘DstPortNumber’.*
- *Concatenate the destination block name to the destination port number and source block name to the source port number*
- *Check if there exists another line that is from the same source block and port but has more destination blocks than the currently considered line.*
- *If another line from the same source block and port with more destinations is detected then skip that line.*

- *Else, perform the following:*
 - *Increment the Saboteur and 'From' block indicator in the name by 1.*
 - *Increment the Position coordinates for the Saboteur and 'From' block to be inserted.*
 - *Add new Saboteur block from the FI library into the selected saboteur position.*
 - *Add new 'From' block from the FI library into the selected 'From' block position.*
 - *Delete line between source port and each of the line's destination ports.*
 - *Add line between source port and newly inserted Saboteur port 2.*
 - *Add line between the newly inserted 'From' block to newly inserted Saboteur port1.*
 - *Add line between the newly inserted Saboteur block port 1 to all the destination ports.*

In this manner, the challenge 3 identified for realizing property based fault injection, listed in section 9.6.1 is addressed. This automated FI framework can be utilized to perform Property based, Random, Strategic and Sequenced Fault injection. Random Fault injection can be implemented by using a Matlab code or Simulink subsystem to randomly select fault location values from the entire range of saboteur identifiers inserted in the model. The fault type and other fault parameters can also be randomly chosen from the set of pre-defined valid values. Strategic fault injection can be conducted by running the Simulink behavioral models with input vectors and then strategically injecting faults of specific type into specific locations at the desired time during the simulation. By varying the FI control signals using Signal builder blocks during the simulation, a sequence of faults can also be injected into the model one after the other. The FI framework is as of now only capable of injecting fault in a single fault location in a clock cycle.

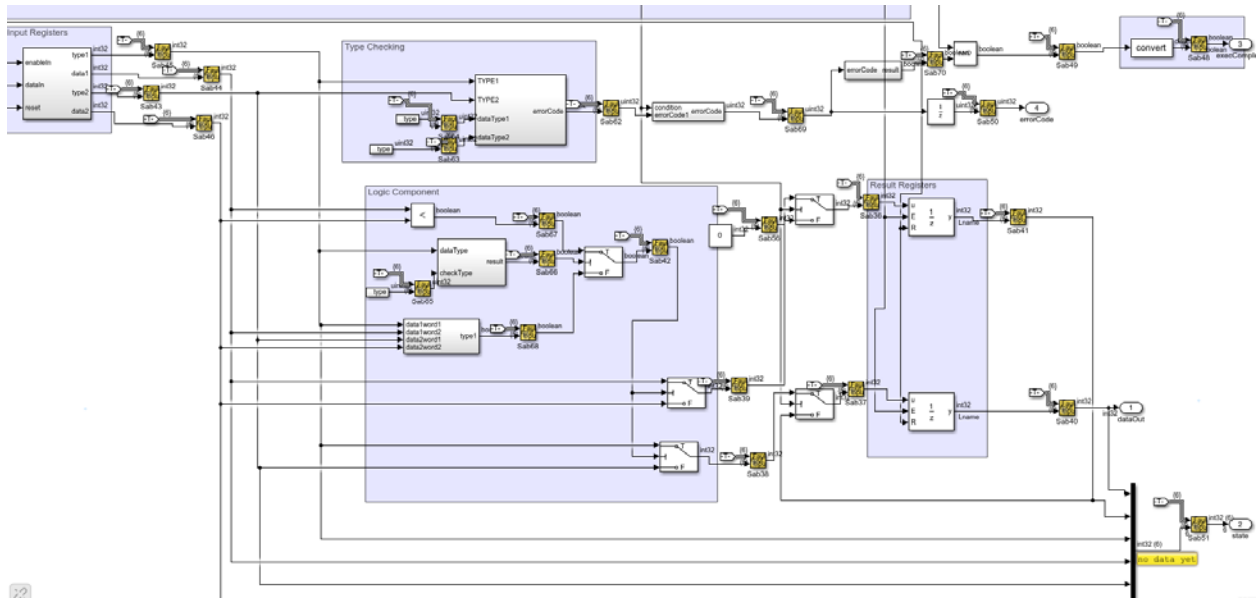


Figure 63: Saboteur Inserted model

9.8 Application of Property Based Fault Injection

This section gives few examples of applying the property based fault injection on SymPLe architecture, and highlights the results obtained during the process. The representative fault tolerant system we evaluated for fault injection experiments is the function blocks in SymPLe architecture [51]. Function blocks are the critical components of SymPLe architecture. SymPLe function Blocks employ several fault tolerance techniques to ensure the safety and reliability of applications executing on SymPLe. Hardware redundancy is one of the main fault tolerance strategy implemented at the function block level. As shown in Figure 64, each FB is constructed as a duplex system as a means of detecting single and multi-event upsets. The inputs, outputs and states of the individual function blocks are compared to report data mismatch error. Any faults within individual function blocks if propagated to the outputs are caught by the state comparator which compares the duplicated function blocks and reports a State error.

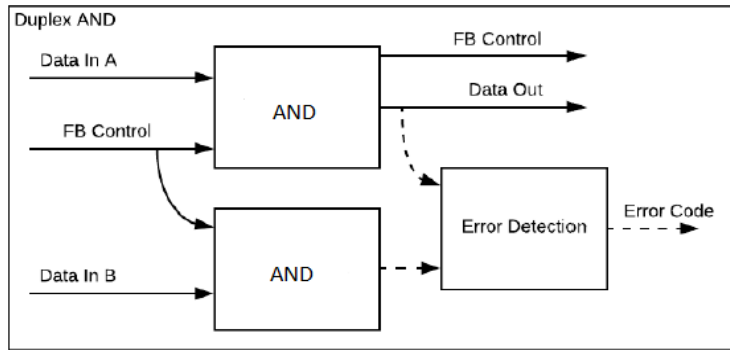


Figure 64: Hardware Redundancy in Function Blocks [48]

The fault aware SymPLe function blocks detect errors in SymPLe's control flow execution and data path execution. There are basically two classes of errors that are detected; (1) execution errors, and (2) computation errors. An execution error is an error that violates the execution semantics of SymPLe computational model. A computation error is an error that affects logic and arithmetic operations. All the error detection capabilities of function blocks can be seen in Table 5. Prolonged execution of a function block could be a sign of malfunction within the function blocks due to transient or permanent physical faults. This execution error is detected and reported using the execution timeout feature of the function block controller. SymPLe is designed to detect and tolerate transient and permanent faults. Function blocks recover from transient errors by retrying the execution. Non transient faults result in several unsuccessful execution retries. This will cause global sequencer to issue a fail-stop signal state where

SymPLe task lane will cease execution before the error propagates to the system outputs. These fault detection/tolerance features constitute a fail-fast/fail-stop failure semantics for the function blocks.

Table 5: Function Block Error Detection Capabilities [51]

| Error Type | Error Name | Variable Name | Value | Binary Value |
|-------------|-----------------------------------|-------------------|-------|--------------|
| Computation | Duplex Divergence Error* | stateErrorBit | 1 | 0b1 |
| Execution | Function Block Execution Timeout* | timeoutBit | 2 | 0b10 |
| Computation | Arithmetic Overflow | overflowBit | 4 | 0b100 |
| Computation | Arithmetic Underflow | underflowBit | 8 | 0b1000 |
| Computation | Arithmetic Division by 0 | divideByZeroBit | 16 | 0b10000 |
| Execution | Input Register Read Overflow | inputOverflowBit | 32 | 0b100000 |
| Execution | Output Register Write Overflow | outputOverflowBit | 64 | 0b1000000 |
| Execution | Data Type Error* | typeBit | 256 | 0b100000000 |

9.8.1 Use case 1- Verifying Failure Semantics of SymPLe Function Blocks

The first use case of model-based fault injection is to verify “the fail-stop/Fail fast” semantics of the duplex redundancy feature in the function blocks. The first step to conducting the model based fault injection experiments is to instrument the individual function blocks with saboteurs in all the lines, in order to give the capability to inject different types of faults on any signal within the individual function blocks. The Fault injection control signals are accessible to the user as external inputs. So both classical fault injection testing and Property proving based fault injection is feasible on the model.

Property Proof and Assumptions:

When Property proving method is adopted, fault tolerance properties to be always held true during the execution is modeled using Simulink Design verifier blocks. For verifying the hardware redundancy feature, the critical property to be considered is that the state comparator detects anomalies and sends a high on ‘state_error’ signal when the function blocks in the duplex configuration hold different states. The

property for verifying the hardware redundancy feature of the ‘ADD’ function block in SymPLe architecture is modeled as shown in Figure 65. The mismatch property states that:

“a difference between the states of the ADD1 and ADD2 function blocks, implies that the ‘state_error’ signal is true”.

The faults injected are activated when they manifest into errors in the system. The faults injected in function blocks gets activated when erroneous values propagate to the outputs of the function blocks causing a mismatch between the two function blocks. For each fault location, different preconditions are applicable to ensure the fault is activated and actually propagates far enough to cause a failure at the output. So to direct the model checker to consider input conditions that can activate the faults, an assumption condition as stated below is modeled as shown in Figure 65.

“there will be mismatch between the states (which comprises of the inputs read into FBs and FB outputs) of the ADD1 and ADD2 function blocks, within 5 cycles after a fault is injected”

With this assumption in place, the Design verifier explores the entire input space and finds all possible input sequences that can activate a specific fault in the model. Thus Design Verifier covers the entire fault activation space from within the total input space, during property verification.

Another important assumption to be made is that the *“redundant data inputs to the two function blocks are always equal”*, as shown in Figure 66. This is to ensure that the state mismatch between the duplex function blocks is not caused due to the mismatch of inputs to the function blocks, but instead caused due to the fault injected within one of the function blocks.

Figure 68 shows assumptions on the Fault Injection Control signals to constrain the fault space to a representative set of fault locations and fault models during the FI experiment. The fault location is assumed to be between two parameter values ‘faultlocationmin’ and ‘faultlocationmax’, thereby enabling Design Verifier to cover all fault locations between the given range. The ‘FaultType’ is assumed to be any values 1 to 5, so that only all the defined fault models are considered for property proving. ‘FaultDuration’ parameter is left open with no assumptions so that the Design Verifier can cover the entire datatype range of the parameter for proving the safety property. No specific assumption on the ‘BitNum’ signal enables Design Verifier to consider all possible single and multi-bit faults. The Injection Time parameter (‘InjectFault’ signal) is also left open without any constraints, for the Design Verifier to validate the property by exploring or considering different injection times.

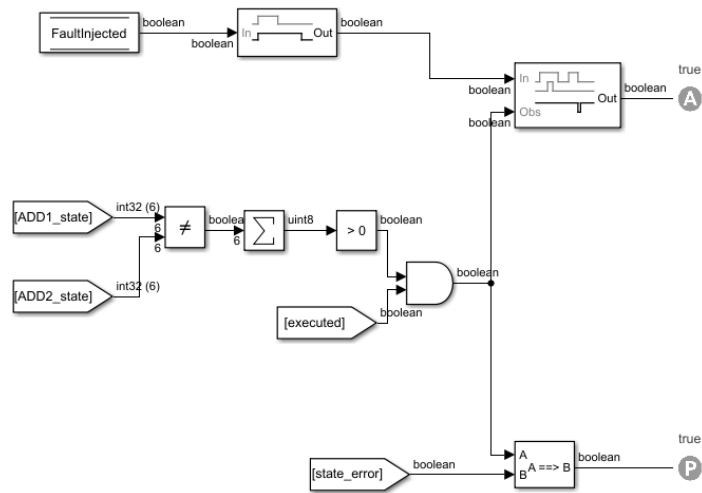


Figure 65: Property for proving hardware redundancy of Function Blocks

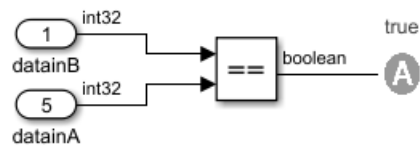


Figure 66: Input data equivalence



Figure 67: Funcblock selection

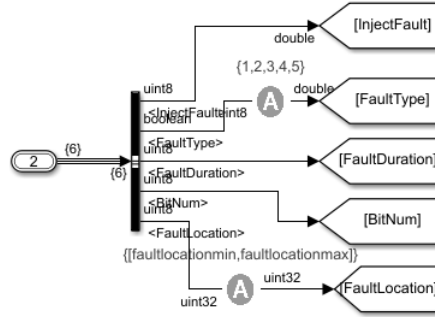


Figure 68: Fault Injection Control Signals

Results:

Fault campaign was run by varying the fault locations (Figure 68) and ‘funcblock’ parameter (Figure 67) and checking for obedience to the ‘State mismatch detection’ safety property. This way, the complete fault space for the Function block array in SymPLe architecture is covered. The FI campaign is automated by setting the necessary parameters and invoking the Simulink design verifier for property proving, repeatedly from within a Matlab code. The summary of the fault campaign on 6 different function blocks are provided in Table 6. The results indicate that faults injected on any of the function blocks in the duplex configuration satisfied the safety property as it always resulted in state mismatch error being detected by the system by pulling the ‘state_error’ signal high. But any faults within the State Comparators in the function blocks caused the safety property to be violated. State comparators were identified as the single point of failures within the function block design. These results are reasonable with hindsight, as state comparators are a vital circuit in the SymPLe function block architecture. The state comparator fault tolerance capabilities were overlooked in the design.

Table 6: Function Block Fault Injection Campaign Results

| Fault Location IDs | Total No: Fault Locations | Faults in block | Function Block | Fault Type | Property | Time taken (mins) | Proof Validity |
|--------------------|---------------------------|----------------------|----------------|------------------------------------|--------------------------|-------------------|----------------|
| 485-520 | 36 | AND1 | AND | Transient Multi or Single Bit Flip | State Mismatch Detection | 15.3 | Valid |
| 449-484 | 36 | AND2 | AND | Transient Multi or Single Bit Flip | State Mismatch Detection | 16.00 | Valid |
| 521-528 | 8 | AND State Comparator | AND | Transient Multi or Single Bit Flip | State Mismatch Detection | 17.3 | Falsified |
| 405-440 | 36 | OR1 | OR | Transient Multi or | State | 14:09 | Valid |

| | | | | | | | |
|---------|----|----------------------|-----|------------------------------------|--------------------------|-------|-----------|
| | | | | Single Bit Flip | Mismatch Detection | | |
| 369-404 | 36 | OR2 | OR | Transient Multi or Single Bit Flip | State Mismatch Detection | 14.37 | Valid |
| 441-448 | 8 | OR State Comparator | OR | Transient Multi or Single Bit Flip | State Mismatch Detection | 16.06 | Falsified |
| 220-252 | 33 | MAX1 | MAX | Transient Multi or Single Bit Flip | State Mismatch Detection | 11.17 | Valid |
| 187-219 | 33 | MAX2 | MAX | Transient Multi or Single Bit Flip | State Mismatch Detection | 11.51 | Valid |
| 253-260 | 8 | MAX State Comparator | MAX | Transient Multi or Single Bit Flip | State Mismatch Detection | 17.25 | Falsified |
| 36-70 | 35 | MIN1 | MIN | Transient Multi or Single Bit Flip | State Mismatch Detection | 16.01 | Valid |
| 1-35 | 35 | MIN2 | MIN | Transient Multi or Single Bit Flip | State Mismatch Detection | 15.12 | Valid |
| 71-78 | 8 | MIN State Comparator | MIN | Transient Multi or Single Bit Flip | State Mismatch Detection | 17.53 | Falsified |
| 129-178 | 50 | ADD1 | ADD | Transient Multi or Single Bit Flip | State Mismatch Detection | 26.16 | Valid |
| 79-128 | 50 | ADD2 | ADD | Transient Multi or Single Bit Flip | State Mismatch Detection | 26.24 | Valid |
| 179-186 | 8 | ADD State Comparator | ADD | Transient Multi or Single Bit Flip | State Mismatch Detection | 15.43 | Falsified |
| 261-310 | 50 | SUB1 | SUB | Transient Multi or Single Bit Flip | State Mismatch Detection | 26.36 | Valid |
| 311-360 | 50 | SUB2 | SUB | Transient Multi or Single Bit Flip | State Mismatch Detection | 28.51 | Valid |
| 361-368 | 8 | SUB State Comparator | SUB | Transient Multi or Single Bit Flip | State Mismatch Detection | 19.3 | Falsified |

9.8.2 Use case 2 - Verifying Timeout functionality during SymPLe Function Blocks Execution

Another example that shows the usage of property proof based fault injection is given below. A critical safety feature within SymPLe architecture is its detection of malfunction of the system by detecting a prolonged function block execution. The function block execution is expected to timeout, send an error and restart the task, when the execution of a function block extends beyond 50 cycles. The second use case of property-based fault injection is to verify this execution timeout feature of function blocks.

Property Proof and Assumptions:

For verifying the function block execution timeout feature, the critical property that is expected to hold true is that:

“When ‘execute’ signal is detected high for 50 cycles, the ‘error’ signal and timeout bit in the error code shall be set to true indicating the timeout error detection.”

This property is modeled as shown in Figure 69. To verify the property a fault has to be injected that can cause function block execution to timeout and cause an error. This error condition is modeled as an assumption as given in Figure 69. It states that

“Once ‘execute’ signal has gone high and execution has started, within next 50 cycles execution shall not be completed and ‘executed’ signal should not become high”.

This assumption causes Design Verifier to explore the design and identify all possible faults and input conditions to activate these faults. One direct example of a fault that can cause timeout error is the ‘stuck at 0’ fault (in bit 0) on the boolean ‘executed’ signal output from the function blocks. Similarly, there could be several faults that can create a delay in execution.

This example also depicts the complete automation of fault and input space exploration by Design Verifier with minimal specification from the tester on the error scenario and fault tolerance property.

Results:

The time-out property was validated true, indicating that a prolonged execution of function block extending beyond 50 cycles was detected and signaled as an error. A property proving experiment, assuming a single fault type (stuck-at-0) and all fault locations in a single function block, took ~50 minutes to complete the time-out property verification.

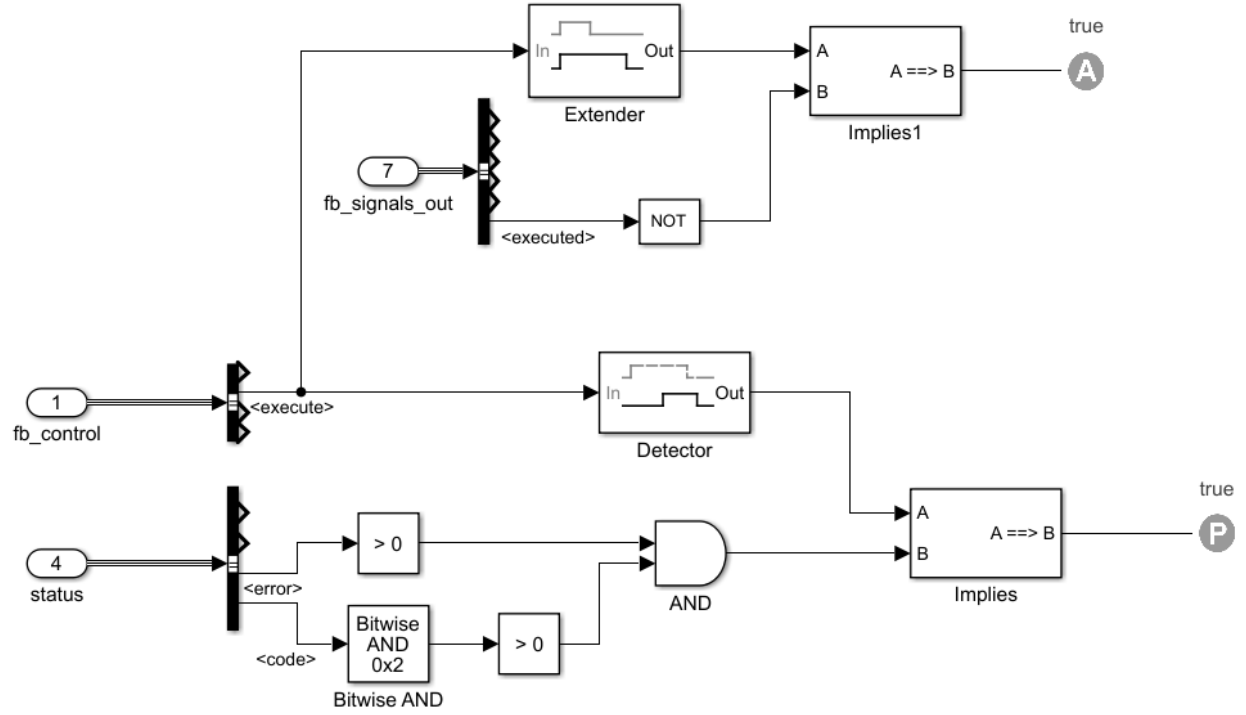


Figure 69: Timeout Detection Feature Property

9.8.3 Efficiency Comparison between Classical and Property-Based FI

As the Use Case 1 results (Table 6) indicate, the property based fault injection exhaustively covers the entire input, state and fault space for a single fault location in approximately 30 secs. This is huge improvement compared to classical fault injection which would otherwise take several thousands of manual experiments to cover the same input, state and fault space.

To quantitatively measure the efficiency of property based fault injection with respect to Classical fault injection, we compare the time taken to cover the fault space for a single fault location within the model with both these methods. The fault locations within the model are signal lines within the model. Here we consider a signal of uint32 datatype. Multiple factors need to be considered to calculate the applicable fault space for a signal or fault location within the model. Here we restrict the fault space measurement to a single fault type, eg: Transient bit flip fault.

For Transient bit flip fault type, there can be single or multi bit flips affecting 1 to 32 bits of the signal, resulting in 2^{32} different bit flip possibilities. Additionally, considering the fault duration factor, each

faulty value can last for 1 to 255 clock cycles. Hence each faulty value needs to be considered for 255 fault duration values, thus making the fault space to $2^{32} * 255$. Furthermore, each of the 2^{32} possible input values for the signal can have $(2^{32} * 255)$ possibilities of bit flips. Hence the

Total Experiment space for Transient bit flip fault for a single fault location = $2^{32} * 2^{32} * 255 = 2^{64} * 255 = 4.7e+21$

Considering the current state of the practice in simulation based Fault injection, there are fault simulators which have ability to inject a fault and store the state as fast 100 μ sec. By making an optimistic assumption that it takes ~ 100 μ secs to run a single classical fault injection test, it will take $1.31e+14$ hrs to cover the entire FI experiment space for a single fault location.

$$4.7e+21 * 0.0001 \text{ secs} = 4.70e+17 \text{ secs} = 1.31e+14 \text{ hrs}$$

On the other hand, our experiments with Simulink Design verifier and results (in Table 6) indicate that Property based fault injection method takes an extremely short time of ~ 30 secs to validate a safety property for a single fault location, by completely covering the same vast experiment space. This shows the drastic improvement in Fault injection efficiency/productivity achieved with Property Based Fault injection as compared to Classical Fault injection.

Chapter 10 Conclusions and Future Work

The first part of the thesis addresses the challenges of design assurance for cyber-physical systems from a model-based design and testing perspective. An end-to-end model-based design assurance workflow that can be applied to safety-critical systems aimed at IEC 61508 certification was developed, implemented and assessed. The results presented strongly suggest the effectiveness of Model-Based Design Assurance in the identification of design flaws early in the product lifecycle. This work highlights the importance of systematic model testing in guiding the subsequent verification steps like Property proving, Code and Hardware verification. The discovered synergy between model-testing and formal verification had a significant value in the design assurance process as it demonstrated capability to help find difficult design flaws. The second part of the thesis focuses on assessing the fault tolerance aspects of a FPGA-based system in the presence of physical faults. This thesis introduces and develops a novel fault injection framework that utilizes the benefits of model checking and model-based design to achieve an efficient exploration of representative fault space to achieve near exhaustive fault injection. A new class of saboteur fault injection modules were developed for Simulink based models. The saboteurs developed can encompass multiple fault models based on physical faults in FPGA-based systems or other hardware-based systems. The property based-fault injection ensures a complete coverage of the fault space and automated means to identify the fault activation space. The time savings as compared to traditional/classical fault injection is orders of magnitude better efficiency.

For the future, it is planned to exercise the proposed model based design assurance workflow on microcontroller based systems instead of FPGA-based systems and study the differences between HDL workflow and C-language workflow. It would also be of significant value if a well-defined methodology and process is formulated for the systematic application of the discovered inherent synergy between model testing and property proving in the design assurance process of safety critical systems, to improve the overall efficiency of formal verification phase. It is also in the future plan to study the proposed property based fault injection framework with respect to hardware-level fault injection to see if they are equivalent. It would be interesting to analyze if the same benefits of the property based FI at the model level refine and hold true at the hardware level also and identify any additional challenges that arise when trying to apply the method at hardware level. Last but not the least, it would be beneficial to assess the fault coverage of this formal FI approach to quantitatively guarantee the exhaustiveness of the input, state and fault space coverage.

References

- [1] James Manyika *et al.*, “THE INTERNET OF THINGS: MAPPING THE VALUE BEYOND THE HYPE,” Mckinsey Global Institute, Jun. 2015. Accessed: Apr. 06, 2020. [Online]. Available: <https://www.telecomcircle.com/wp-content/uploads/2018/05/Mckinsey-Report-on-IoT-Mapping-the-value-beyond-the-hype.pdf>.
- [2] B. W. Johnson, *Design and Analysis of Fault-tolerant Digital Systems*. Addison-Wesley Publishing Company, 1989.
- [3] “April 2014 Multistate 911 Outage: Cause and Impact Report and Recommendations,” Public Safety and Homeland Security Bureau Federal Communications Commission, Public Safety Docket No. 14-72 PSHSB Case File Nos. 14-CCR-0001-0007, Oct. 2014.
- [4] Democratic Staff of the and House Committee on Transportation and Infrastructure, “TI Preliminary Investigative Findings Boeing 737 MAX March 2020.pdf,” Mar. 2020. Accessed: Apr. 06, 2020. [Online]. Available: <https://transportation.house.gov/imo/media/doc/TI%20Preliminary%20Investigative%20Findings%20Boeing%20737%20MAX%20March%202020.pdf>.
- [5] N. Leveson, *Engineering a safer world: systems thinking applied to safety*. Cambridge, Mass.: The MIT Press, 2012.
- [6] M. P. E. Heimdahl, “Safety and Software Intensive Systems: Challenges Old and New,” in *Future of Software Engineering (FOSE '07)*, May 2007, pp. 137–152, doi: 10.1109/FOSE.2007.18.
- [7] M. Saravi, L. Newnes, A. R. Mileham, and Y. M. Goh, “Estimating Cost at the Conceptual Design Stage to Optimize Design in terms of Performance and Cost,” in *Collaborative Product and Service Life Cycle Management for a Sustainable World*, R. Curran, S.-Y. Chou, and A. Trappey, Eds. London: Springer London, 2008, pp. 123–130.
- [8] Frola, F and C. Miller, “System safety in aircraft management,” Logistics Management Institute, Washington DC, 1984.
- [9] Adam Strafaci, “What does BIM mean for civil engineers,” *CE News, Transportation* 127, 2008.
- [10] “IEC_61508-3.pdf.”
- [11] C. Perrow, *Normal accidents: Living with high risk technologies*. Princeton University Press, 2011.
- [12] J.-C. Laprie, “Dependability of computer systems: concepts, limits, improvements,” in *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE '95*, Oct. 1995, pp. 2–11, doi: 10.1109/ISSRE.1995.497638.
- [13] A. Kornecki and J. Zalewski, “Certification of software for real-time safety-critical systems: state of the art,” *Innovations Syst Softw Eng*, vol. 5, no. 2, pp. 149–161, Jun. 2009, doi: 10.1007/s11334-009-0088-1.
- [14] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan. 2004, doi: 10.1109/TDSC.2004.2.
- [15] J.-C. Laprie and B. Randell, “Fundamental Concepts of Dependability,” University of Newcastle upon Tyne, Computing Science, 2001.
- [16] Yangyang Yu and B. W. Johnson, “Safety assessment for safety-critical systems including physical faults and design faults,” in *RAMS '06. Annual Reliability and Maintainability Symposium, 2006.*, Jan. 2006, pp. 588–593, doi: 10.1109/RAMS.2006.1677437.
- [17] “(4) (PDF) Dynamic Partial Reconfiguration of FPGA for SEU Mitigation and Area Efficiency.” https://www.researchgate.net/publication/261177496_Dynamic_Partial_Reconfiguration_of_FPGA_for_SEU_Mitigation_and_Area_Efficiency (accessed Jan. 30, 2020).
- [18] R. C. Baumann, “Radiation-induced soft errors in advanced semiconductor technologies,” *IEEE Trans. Device Mater. Reliab.*, vol. 5, no. 3, pp. 305–316, Sep. 2005, doi: 10.1109/TDMR.2005.853449.
- [19] “Introduction to Single-Event Upsets,” p. 10, 2013.

- [20] C. R. Elks, *A theory of run-time verification for safety critical reactive systems*. University of Virginia, 2005.
- [21] K. J. Hayhurst, *A practical tutorial on modified condition/decision coverage*. DIANE Publishing, 2001.
- [22] D. R. Kuhn, R. Chandramouli, and R. W. Butler, “Cost Effective Use of Formal Methods in Verification and Validation,” p. 38.
- [23] B. W. Johnson, “An Introduction to the Design and Analysis of Fault-Tolerant Systems,” p. 109.
- [24] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*. Elsevier, 2010.
- [25] Carl R Elks, N. J. George, M. A. Reynolds, M. Miklo, and C. Berger, “Development of a Fault Injection-Based Dependability Assessment Methodology for Digital and I&C Systems,” United States Nuclear Regulatory Commission, Office of Nuclear Regulatory Research, NUREG/CR-7151, 2012.
- [26] H. Ziade, R. A. Ayoubi, and R. Velazco, “A Survey on Fault Injection Techniques,” *Int. Arab J. Inf. Technol.*, vol. 1, pp. 171–186, 2004.
- [27] “QEMU documentation - QEMU.” <https://www.qemu.org/documentation/> (accessed Apr. 11, 2020).
- [28] “OVPSim Simulator.” /technology_ovpsim (accessed Apr. 11, 2020).
- [29] J. Na and D. Lee, “Simulated Fault Injection Using Simulator Modification Technique,” *ETRI Journal*, vol. 33, no. 1, pp. 50–59, 2011, doi: 10.4218/etrij.11.0110.0106.
- [30] K. Goseva-Popstojanova, “Report: Survey on Model-Based Software Engineering and Auto-Generated Code,” p. 56, 2016.
- [31] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, “Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice,” *Softw Syst Model*, vol. 17, no. 1, pp. 91–113, Feb. 2018, doi: 10.1007/s10270-016-0523-3.
- [32] Anitha Murugesan *et al.*, “From Requirements to Code: Model Based Development of a Medical Cyber Physical System,” presented at the International Symposium on Foundations of Health Informatics Engineering and Systems International Workshop on Software Engineering in Health Care, Jul. 2017.
- [33] D. Bhatt, B. Hall, S. Dajani-Brown, S. Hickman, and M. Paulitsch, “Model-based development and the implications to design assurance and certification,” in *24th Digital Avionics Systems Conference*, Oct. 2005, vol. 2, pp. 13 pp. Vol. 2-, doi: 10.1109/DASC.2005.1563401.
- [34] E. Bringmann and A. Krämer, “Model-Based Testing of Automotive Systems,” in *and Validation 2008 1st International Conference on Software Testing, Verification*, Apr. 2008, pp. 485–493, doi: 10.1109/ICST.2008.45.
- [35] M. Beine and D. Fleischer, “A Model-Based Reference Workflow for the Development of Safety-Related Software,” presented at the SAE Convergence 2010, Oct. 2010, pp. 2010-01–2338, doi: 10.4271/2010-01-2338.
- [36] M. Conrad and G. Sandmann, “A Verification and Validation Workflow for IEC 61508 Applications,” presented at the SAE World Congress & Exhibition, Apr. 2009, pp. 2009-01–0271, doi: 10.4271/2009-01-0271.
- [37] M. Conrad, “Verification and Validation According to ISO 26262: A Workflow to Facilitate the Development of High-Integrity Software,” p. 8.
- [38] S. Marrone, F. Flammini, N. Mazzocca, R. Nardone, and V. Vittorini, “Towards Model-Driven V&V assessment of railway control systems,” *Int J Softw Tools Technol Transfer*, vol. 16, no. 6, pp. 669–683, Nov. 2014, doi: 10.1007/s10009-014-0320-7.
- [39] R. Rana, M. Staron, C. Berger, J. Hansson, M. Nilsson, and F. Törner, “Early Verification and Validation According to ISO 26262 by Combining Fault Injection and Mutation Testing,” in *Software Technologies*, vol. 457, J. Cordeiro and M. van Sinderen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 164–179.
- [40] “Entering the Digital Age - Power Engineering.” <https://www.power-eng.com/articles/print/volume-116/issue-9/features/entering-the-digital-age.html> (accessed Jun. 24, 2019).

- [41] E.-S. Kim, D.-A. Lee, S. Jung, J. Yoo, J.-G. Choi, and J.-S. Lee, “NuDE 2.0: A Formal Method-based Software Development, Verification and Safety Analysis Environment for Digital I&Cs in NPPs,” *Journal of Computing Science and Engineering*, vol. 11, no. 1, p. 15, 2017.
- [42] M. Eslami, B. Ghavami, M. Raji, and A. Mahani, “A survey on fault injection methods of digital integrated circuits,” *Integration*, vol. 71, pp. 154–163, Mar. 2020, doi: 10.1016/j.vlsi.2019.11.006.
- [43] Y. S. Jeong, S. M. Lee, and S. E. Lee, “A Survey of Fault-Injection Methodologies for Soft Error Rate Modeling in Systems-on-Chips,” *Bulletin EEI*, vol. 5, no. 2, pp. 169–177, Jun. 2016, doi: 10.11591/eei.v5i2.617.
- [44] S. Bingham and J. Lach, “Enhanced Fault Coverage Analysis Using ABVFI,” p. 6.
- [45] U. Krautz, M. Pflanz, C. Jacobi, H. W. Tast, K. Weber, and H. T. Vierhaus, “Evaluating Coverage of Error Detection Logic for Soft Errors using Formal Methods,” in *Proceedings of the Design Automation Test in Europe Conference*, Mar. 2006, vol. 1, pp. 1–6, doi: 10.1109/DATE.2006.244062.
- [46] R. Leveugle, “A new approach for early dependability evaluation based on formal property checking and controlled mutations,” in *11th IEEE International On-Line Testing Symposium*, Jul. 2005, pp. 260–265, doi: 10.1109/IOLTS.2005.8.
- [47] Daniel Larsson and Reiner Hahnle, “Symbolic Fault Injection,” *International Verification Workshop (VERIFY)*, vol. 259, 2007.
- [48] Rickard Svenningsson, Jonny Vinter, Henrik Eriksson, and Martin Törngren, “MODIFI: a MODEL-implemented fault injection tool,” presented at the International Conference on Computer Safety, Reliability, and Security, Sep. 2010, pp. 210–222.
- [49] M. Moradi, B. V. Acker, and K. Vanherpen, “Model-Implemented Hybrid Fault Injection for Simulink (Tool demonstrations),” p. 16.
- [50] I. Pill, I. Rubil, F. Wotawa, and M. Nica, “SIMULTATE: A Toolset for Fault Injection and Mutation Testing of Simulink Models,” in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Chicago, IL, USA, Apr. 2016, pp. 168–173, doi: 10.1109/ICSTW.2016.21.
- [51] Matt Gibson *et al.*, “Achieving Verifiable and High Integrity Instrumentation and Control Systems through Complexity Awareness and Constrained Design,” Electric Power Research Institute (EPRI), 15–8044, Jul. 2019.
- [52] S. Friedenthal, R. Griego, and M. Sampson, “INCOSE Model Based Systems Engineering (MBSE) Initiative,” p. 30.
- [29] Alfred Helmerich *et al.*, “Study of Worldwide Trends and R&D Programmes in Embedded Systems in View of Maximising the Impact of a Technology Platform in the Area,” FAST GmbH, Munich, Germany, Technische Universität München, Germany, Nov. 2005.
- [54] C. Olosky, “Model Based V and V Workshop.pptx,” presented at the MATHWORKS: Model Based V&V Workshop, Oct. 30, 2018.
- [55] “ANSYS SCADE Suite: Model-Based Development.” <https://www.ansys.com/products/embedded-software/ansys-scade-suite> (accessed Jul. 11, 2019).
- [56] “Questa® Property Checking.” <https://www.mentor.com/products/fv/questa-property-checking> (accessed Mar. 09, 2020).
- [57] M. Boule and Z. Zilic, “Incorporating efficient assertion checkers into hardware emulation,” in *2005 International Conference on Computer Design*, Oct. 2005, pp. 221–228, doi: 10.1109/ICCD.2005.66.
- [58] Carl Elks *et al.*, “Preliminary Results of a Bounded Exhaustive Testing Study for Software in Embedded Digital Devices in Nuclear Power Applications,” Idaho National Laboratory U.S. Department of Energy Office of Nuclear Energy, Sep. 2019. Accessed: Feb. 29, 2020. [Online]. Available: https://lwrs.inl.gov/Advanced%20IIC%20System%20Technologies/Preliminary_Results_Bounded_Exhaustive_Testing_Study_Software_Embedded_Digital_Devices_NP_Applications.pdf.

- [59] W. J. Aldrich, "Using Model Coverage Analysis to Improve the Controls Development Process," 2002, doi: 10.2514/6.2002-4684.
- [60] Mathworks, "Types of Model Coverage - MATLAB & Simulink," *Types of Model Coverage*. <https://www.mathworks.com/help/slcoverage/ug/types-of-model-coverage.html> (accessed Jun. 06, 2019).
- [61] D. Smith, T. DeLong, and B.W. Johnson, "A Safety Assessment Methodology for Complex Safety Critical Hardware/Software Systems.," presented at the International Topical Meeting on Nuclear Plant Instrumentation, Controls, and Human-Machine Interface Technology, Washington DC, 2000.
- [62] L. Kaufman and B.W. Johnson, "The Importance of Fault Detection Coverage in Safety Critical Systems," in *NUREG/CP-0166, Proceedings of the 26th Water Reactor Safety Information Meeting: U.S. NRC*, 1998.
- [63] C. Elks, B.W. Johnson, and M. Reynolds, "A Perspective on Fault Injection Methods for Nuclear Safety Related Digital I&C Systems," presented at the 6th International Topical Meeting on Nuclear Plant Instrumentation Control and Human Machine Interface Technology., Knoxville, TN: NPIC&HMIT, 2009.
- [64] C. Elks, M. Reynolds, B. Johnson, N. George, M. Waterman, and J. Dion, "Application of a Fault Injection Based Dependability Assessment Process to a Commercial Safety Critical Nuclear Reactor Protection System," presented at the Dependable Systems and Networks Symposium, Chicago, IL, 2010.
- [65] M.Reynolds, C.R. Elks, N. George, M.Sekhar, T. Delong, and B.W. Johnson, "A Quantitative Safety Assessment Methodology for Safety-Critical Programmable Electronic Systems Using Fault Injection," presented at the SAE World Congress., Detroit, MI, 2009.
- [66] T. Aldemir *et al.*, *Dynamic Reliability Modeling of Digital Instrumentation and Control Systems for Nuclear Reactor Probabilistic Risk Assessments*, "NUREG/CR-6942. 2007.
- [67] Bing Huang, Xiaojun Li, Ming Li, J. Bernstein, and C. Smidts, "Study of the Impact of Hardware Fault on Software Reliability," in *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, Chicago, IL, USA, 2005, pp. 63–72, doi: 10.1109/ISSRE.2005.39.
- [68] B. Huang, M. Rodriguez, M. Li, J. B. Bernstein, and C. S. Smidts, "Hardware Error Likelihood Induced by the Operation of Software," *IEEE Transactions on Reliability*, vol. 60, no. 3, pp. 622–639, Sep. 2011, doi: 10.1109/TR.2011.2161699.
- [69] L. Chu, G. Martinez-Gridi, M. Yue, J. Lehner, and P. Samanta, "Traditional Probablistic Risk Assessment Methods for Digital Systems," NUREG/CR-6962, NRC, 2008. Accessed: Apr. 13, 2020. [Online]. Available: <https://www.nrc.gov/docs/ML0831/ML083110448.pdf>.
- [70] Arlat Jean, Yves Crouzet, and Jean Claudie Raprie, "Fault Injection for the Experimental Validation of Fault Tolerance," presented at the Proc. Esprit Conference, 1991.
- [71] J. Arlat *et al.*, "Fault injection for dependability validation: a methodology and some applications," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166–182, Feb. 1990, doi: 10.1109/32.44380.
- [72] Yangyang Yu, B. Bastien, and B. W. Johnson, "A state of research review on fault injection techniques and a case study," in *Annual Reliability and Maintainability Symposium, 2005. Proceedings.*, Alexandria, VA, USA, 2005, pp. 386–392, doi: 10.1109/RAMS.2005.1408393.
- [73] J. Arlat, A. Costes, Y. Crouzet, J. C. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 913–923, Aug. 1993, doi: 10.1109/12.238482.
- [74] Y. Yu and B. W. Johnson, "Fault Injection Techniques," in *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, A. Benso and P. Prinetto, Eds. Boston, MA: Springer US, 2003, pp. 7–39.
- [75] A. Benso and P. Prinetto, *Fault Injection Techniques and Tools for Embedded Systems*. Secaucus, UNITED STATES: Kluwer Academic Publishers, 2003.
- [76] R. Barbosa, J. Vinter, P. Folkesson, and J. Karlsson, "Assembly-Level Pre-injection Analysis for Improving Fault Injection Efficiency," in *Dependable Computing - EDCC 5*, vol. 3463, M. Dal Cin,

- M. Kaâniche, and A. Pataricza, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 246–262.
- [77] Pescosolido, M, “Statistical Models for Coverage Estimation.,” School of Engineering and Applied Science Masters Thesis, University of Virginia, 2002.
 - [78] D. T. Smith, B. W. Johnson, and J. A. Profeta, “System dependability evaluation via a fault list generation algorithm,” *IEEE Transactions on Computers*, vol. 45, no. 8, pp. 974–979, Aug. 1996, doi: 10.1109/12.536240.
 - [79] H. Alemzadeh *et al.*, “Systems-Theoretic Safety Assessment of Robotic Telesurgical Systems,” in *Computer Safety, Reliability, and Security*, Cham, 2015, pp. 213–227.
 - [80] P. Sun, L. Garcia, and S. Zonouz, “Tell Me More Than Just Assembly! Reversing Cyber-Physical Execution Semantics of Embedded IoT Controller Software Binaries,” in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Portland, OR, USA, Jun. 2019, pp. 349–361, doi: 10.1109/DSN.2019.00045.
 - [81] S. Jha *et al.*, “ML-Based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection,” in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Portland, OR, USA, Jun. 2019, pp. 112–124, doi: 10.1109/DSN.2019.00025.
 - [82] C. Baier and J.-P. Katoen, *Principles of model checking*. Cambridge, Mass: The MIT Press, 2008.
 - [83] V. Delebarre and J.-F. Etienne, “Proving Global Properties with the Aid of the SIMULINK DESIGN VERIFIER Proof Tool,” *Formal Methods: Industrial Use from Model to the Code*, pp. 183–223, 2013.
 - [84] Hana Chockler, Orna Kupferman, and Moshe Y. Vardi, “Coverage metrics for temporal logic model checking,” presented at the International Conference on Tools and Algorithms for the Construction and Analysis of Systems., Berlin, Heidelberg, 2001, Accessed: Mar. 28, 2020. [Online]. Available: <https://www.cse.huji.ac.il/~ornak/publications/tacas01.pdf>.
 - [85] O. Kupferman, W. Li, and S. A. Seshia, “A Theory of Mutations with Applications to Vacuity, Coverage, and Fault Tolerance,” in *2008 Formal Methods in Computer-Aided Design*, Portland, OR, USA, Nov. 2008, pp. 1–9, doi: 10.1109/FMCAD.2008.ECP.29.
 - [86] E. Ghassabani, A. Gacek, M. W. Whalen, M. P. E. Heimdahl, and L. Wagner, “Proof-based coverage metrics for formal verification,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana, IL, Oct. 2017, pp. 194–199, doi: 10.1109/ASE.2017.8115632.
 - [87] D. Große, U. Kuhne, and R. Drechsler, “Analyzing Functional Coverage in Bounded Model Checking,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1305–1314, Jul. 2008, doi: 10.1109/TCAD.2008.925790.
 - [88] Hana Chockler, Orna Kupferma, and Moshe Y. Vardi, “Coverage metrics for formal verification,” presented at the Advanced Research Working Conference on Correct Hardware Design and Verification Methods, Berlin, Heidelberg, 2003, Accessed: Mar. 28, 2020. [Online]. Available: <https://www.cse.huji.ac.il/~ornak/publications/charme03a.pdf>.
 - [89] E. M. Clarke, W. Klieber, P. Zuliani, and et al, *Model Checking and the State Explosion Problem*. .
 - [90] Mathworks, “Simulink® Design Verifier Reference,” Sep. 2018. https://www.mathworks.com/help/releases/R2018b/pdf_doc/sldv/sldv_ref.pdf.
 - [91] R. A. Shafik, P. Rosinger, and B. M. Al-Hashimi, “SystemC-Based Minimum Intrusive Fault Injection Technique with Improved Fault Representation,” in *2008 14th IEEE International On-Line Testing Symposium*, Rhodes, Greece, Jul. 2008, pp. 99–104, doi: 10.1109/IOLTS.2008.25.
 - [92] J.C. Baraza, J. Gracia, D. Gil, and P.J. Gil, “Improvement of Fault Injection Techniques Based on VHDL Code Modification,” presented at the Tenth IEEE International High-Level Design Validation and Test Workshop, 2005, Napa Valley, CA, USA, Dec. 2005.
 - [93] T. Riesgo and J. Uceda, “A fault model for VHDL descriptions at the register transfer level,” in *Proceedings EURO-DAC '96. European Design Automation Conference with EURO-VHDL '96 and Exhibition*, Geneva, Switzerland, 1996, pp. 462–467, doi: 10.1109/EURDAC.1996.558244.

Appendix

Saboteur Insertion Script

```
Editor - C:\Users\ayakumaray\Documents\GitHub\Thesis\Saboteur_Insertion.m
1 function [allLineProperties, sourcePortData] = Saboteur_Insertion mdlName)
2 % Injects Fault Model Saboteurs into all signal lines in an input model.
3 % mdlName: .slx model name without extension
4 % Open simulink model temp_mdl.slx
5 FI_sys_mdl = 'temp_mdl.slx';
6 open_system(mdlName);
7 save_system(mdlName, FI_sys_mdl); %copy the model to another model temp_mdl.slx
8 load('FI.mat') %load FI control signals bus datatype.
9 sab_ind_last = 0;
10
11 FI_sys_arr = {'temp_mdl/ADD_DUPLEX/ADD1/'};
12 [row column] = size(FI_sys_arr)
13 for rowind = 1:row
14     for colind = 1:column
15         clear allLines;
16         clear allLineProperties;
17         %read from the array list of subsystems to inject faults.
18         FI_sys = cell2mat(FI_sys_arr(rowind,colind))
19         InjectFaultBN = strcat(FI_sys,'InjectFault');
20         GotoBN = strcat(FI_sys,'Goto');
21         pos_sab = [0, 10, 20, 30];
22         pos_goto = [0, 20, 20, 40];
23         %Insert FaultInject control signals input port connected to a Goto port
24         add_block('FI_Library/InjectFault', InjectFaultBN, 'Position', pos_sab);
25         add_block('FI_Library/Goto', GotoBN, 'Position', pos_goto);
26         add_line(FI_sys, {'InjectFault/1'}, {'Goto/1'}, 'autorouting', 'on');
27 % Parse the entire model and get the list of all the line handles for this model
28 allLines = find_system(FI_sys, 'SearchDepth', 1, 'FindAll', 'On', 'type', 'line')
29
30 disp(allLines);
31 if (isempty(allLines))
32     return;
33 end
34 % Initialize Line Information Class
35 allLineProperties = LineInformation;
36 Fromstr = "From";
37 Sabstr = "Sab";
38 sab_ind= 0+sab_ind_last;
39 sab_x = 10;
40
41 % Parse through the list of all lines and extract the line properties 'SrcBlock', 'DstBlock',
42 %SrcPortNumber' and DstPortNumber'.
43 for i = 1: length(allLines)
44     allLineProperties(i).Identifier = allLines(i);
45     sourceData = get_param(allLineProperties(i).Identifier, 'SrcBlockHandle');
46     destinationData = get_param(allLineProperties(i).Identifier, 'DstBlockHandle');
47     sourcePortData = get_param(allLineProperties(i).Identifier, 'SrcportHandle');
48     destinationPortData = get_param(allLineProperties(i).Identifier, 'DstportHandle');
49     allLineProperties(i).SourceBlock = get_param(sourceData, 'Name');
50     allLineProperties(i).DestinationBlock = get_param(destinationData, 'Name');
51     allLineProperties(i).SourcePort = get_param(sourcePortData, 'PortNumber');
52     allLineProperties(i).DestinationPort = get_param(destinationPortData, 'PortNumber');
53 end
54
55 %Concatenate the destination block name to the destination port number
56 %Concatenate the source block name to the source port number
```

```

57
58 % parse through the list of all lines
59 for i = 1: length(allLines)
60     DestBlock = convertCharsToStrings(allLineProperties(i).DestinationBlock) + '/' + convertCharsToStrings(allLineProperties(i).DestinationPort);
61     SrcBlock = convertCharsToStrings(allLineProperties(i).SourceBlock) + '/' + convertCharsToStrings(allLineProperties(i).SourcePort);
62     len = length(DestBlock);
63
64     skip = false;
65     % check for a line from the same source block with more destination blocks.
66     for k = 1: length(allLines)
67         src_block = convertCharsToStrings(allLineProperties(k).SourceBlock) + '/' + convertCharsToStrings(allLineProperties(k).SourcePort);
68         dest_block = convertCharsToStrings(allLineProperties(k).DestinationBlock) + '/' + convertCharsToStrings(allLineProperties(k).DestinationPort);
69
70
71         if length(dest_block) > len
72             for j = 1: length(dest_block)
73                 %If another line from same source block with more destinations
74                 %detected then skip that line.
75                 if (dest_block(j) == DestBlock(len)) && (src_block == SrcBlock)
76                     skip = true;
77                     break;
78                 end
79             end
80         end
81     end
82
83 if ((SrcBlock == 'InjectFault/1') || (SrcBlock == 'fb_control/1'))
84     skip = true;
85 end

86
87 %For all unskipped lines, perform the following.
88 if skip == false
89     %Increment the saboteur and from block indicator in the name by 1.
90     sab_ind = sab_ind + 1;
91     FromName = Fromstr + num2str(sab_ind)
92     FromDest = strcat(FI_sys, FromName)
93     FromName = FromName + '/1'
94     SabName = Sabstr + num2str(sab_ind)
95     SabDest = strcat(FI_sys, SabName)
96
97     %Increment the position coordinates for the saboteur and from block.
98     pos_sab = [sab_x, 10, sab_x+20, 30];
99     pos_from = [sab_x, 50, sab_x+20, 60];
100     sab_x = sab_x + 40;
101     % add new Saboteur block from library into the selected saboteur position.
102     add_block('FI_Library/Saboteur', SabDest, 'Position', pos_sab);
103     set_param(SabDest, 'Faultlocation', num2str(sab_ind))
104     % add new From block from library into the selected From block position.
105     add_block('FI_Library/From', FromDest, 'Position', pos_from);
106     SabNamePort1 = SabName + '/1'
107     SabNamePort2 = SabName + '/2'
108     % Delete line between source port and each of its destination ports.
109     for j = 1: length(DestBlock)
110         delete_line(FI_sys, SrcBlock, DestBlock(j));
111     end
112     % Add line between source port and newly inserted Saboteur port2.
113     add_line(FI_sys, (SrcBlock), (SabNamePort2), 'autorouting', 'on');
114     % Add line between the newly inserted from block to newly inserted Saboteur port1.
115     add_line(FI_sys, (FromName), (SabNamePort1), 'autorouting', 'on');

116
117     % Add line between the newly inserted Saboteur block port 1 to all the destination ports.
118     for j = 1: length(DestBlock)
119         add_line(FI_sys, (SabNamePort1), (DestBlock(j)), 'autorouting', 'on');
120     end
121
122 end
123
124 sab_ind_last = sab_ind;
125 end
126
127 save_system(FI_sys_mdl);
128

```